ISSN 0926-5473

IFIP Transactions A-50

PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES

M. COSNARD
G.R. GAO
G.M. SILBERMAN
Editors

IFIP

NORTH-HOLLAND

0

PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES

N00014-94-1-0198

IFIP Transactions A: Computer Science and Technology

International Federation for Information Processing

Technical Committees:

Software: Theory and Practice (TC2)
Education (TC3)
System Modelling and Optimization (TC7)
Information Systems (TC8)
Relationship between Computers and Society (TC9)
Computer Systems Technology (TC10)
Security and Protection in Information Processing Systems (TC11)
Artificial Intelligence (TC12)
Human-Computer Interaction (TC13)
Foundations of Computer Science (SG14)

IFIP Transactions Editorial Policy Board

The IFIP Transactions Editorial Policy Board is responsible for the overall scientific quality of the IFIP Transactions through a stringent review and selection process.

IFIP Transactions Abstracted/Indexed in:
INSPEC Information Services
Index to Scientific & Technical Proceedings®
CompuMath Citation Index®, Research Alert™, and SciSearch®

A-50

PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES

Proceedings of the IFIP WG10.3 Working Conference on
Parallel Architectures and Compilation Techniques, PACT '94
Montréal, Canada, 24-26 August, 1994

Edited by

MICHEL COSNARD
Ecole Normale Superiéure de Lyon
Laboratoire de l'Informatique du Parallélisme
Lyon, France

GUANG R. GAO
McGill University
School of Computer Science
Montréal, Canada

GABRIEL M. SILBERMAN
IBM - T.J. Watson Research Center
Yorktown Heights, NY, U.S.A.

NH

1994

NORTH-HOLLAND
AMSTERDAM • LONDON • NEW YORK • TOKYO

ELSEVIER SCIENCE B.V.
Sara Burgerhartstraat 25
P.O. Box 211, 1000 AE Amsterdam, The Netherlands

Keywords are chosen from the ACM Computing Reviews Classification System, ©1991, with permission.
Details of the full classification system are available from
ACM, 11 West 42nd St., New York, NY 10036, USA.

ISBN: 0 444 81926 6
ISSN: 0926-5473

No responsibility is assumed by the publisher or by IFIP for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions or ideas contained in the material herein.

This book is printed on acid-free paper.

Printed in The Netherlands

PREFACE

As evidenced by the success of the first Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, last year in Orlando, and the superb program of this year's conference, fine and medium grain parallelism is a vital, vibrant research topic. Within this area of research, new developments in superscalar and VLIW architectures, and their associated compilation techniques, have provided exciting new avenues to extract performance from a slowing technology curve.

The second conference in this series, the International Conference on Parallel Architectures and Compilation Techniques- PACT '94- has set as its goal to become the premier forum for researchers, developers and practitioners in the fields of fine and medium grain parallelism. This will establish the yearly conference as one of the main events in the field, and its proceedings as an invaluable archival publication of important results in the area.

This year, the program consists of 28 full-length papers and and a number of short (poster) papers, carefully selected by the Program Committee and external reviewers from among 101 submissions. A keynote speech by Arvind will open the conference, and Jack Dennis will also address the banquet attendees. Bob Rau and Anoop Gupta will present invited talks to open the second and third days of the conference, respectively. The panel session, on the subject of programming multithreaded machines, has been organized by A.P. Wim Bohm. Preceding the conference itself are three tutorial sessions, "Parallelizing Compilers: How to Discover and When to Exploit Parallelism Automatically," by Constantine Polychronopoulos and Alex Nicolau, "Data-Driven and Multithreaded Architectures for High-Performance Computing," by Jean-Luc Gaudiot, and "Mechanisms for Exploiting Instruction Level Parallelism," by Yale Patt. All these make up a high quality, exciting technical program, complemented by the location of the conference in the beautiful city of Montreal.

We are grateful to all those who contributed their time and effort in making PACT '94 a reality. Our Steering Committee members, Michel Cosnard (also Publication chair), Kemal Ebcioglu and Jean-Luc Gaudiot, provided a vital link to the original intent of the conference. Efforts by Herbert Hum (Local Arrangements) Zary Segall (Publicity), Walid Najjar (Treasurer, Registration) are greatly appreciated, as are those of Erik Altman (Graphics, PC Administration). We also acknowledge the members of the Program Committee and reviewers, listed separately in these proceedings.

Finally, we acknowledge the support of our sponsors, the International Federation of Information Processing (IFIP) Working Group 10.3 (Concurrent Systems), and the Association for Computing Machinery (ACM) Special Interest Group on Computer Architecture (SIGARCH), in cooperation with Centre de Recherche Informatique de Montréal (CRIM), the Institute of Electrical & Electronics Engineers (IEEE) Computer Society Technical Committees on Computer Architecture (TCCA) and Parallel Processing (TCPP), the ACM Special Interest Group on Programming Languages (SIGPLAN), and financial support from the Office of Naval Research (ONR), IBM Canada's Center for Advanced Studies (CAS), and Hydro Québec.

Gabriel Silberman
IBM T.J. Watson Research Center
General Chairman

Guang Gao
Mc Gill University
Program Committee Chairman

PROGRAM COMMITTEE

Program Committee Chair:

Prof. Guang R. Gao (*McGill University*, Canada)

Program Committee:

Prof. M. Amamiya (*Kyushu University*, Japan)
Prof. L. Bic (*University of California, Irvine*, USA)
Prof. W. Böhm (*Colorado State University*, USA)
Prof. D. Culler (*University of California, Berkeley*, USA)
Dr. D. DeGroot (*Texas Instruments*, USA)
Dr. K. Ebcioğlu (*IBM*, USA)
Prof. G. Egan (*Swinburne University*, Australia)
Dr. W. O'Farrell (*IBM*, Canada)
Prof. J.-L. Gaudiot (*University of Southern California*, USA)
Prof. W. Giloi (*GMD Institute*, Germany)
Prof. R. Gupta (*University of Pittsburgh*, USA)
Prof. J. Gurd (*University of Manchester*, Britain)
Prof. I. Koren (*University of Massachusetts*, USA)
Prof. M. Lam (*Stanford*, USA)
Prof. B. Lécussan (*ONERA-ENSAE* France)
Prof. E. Lee (*University of California, Berkeley*, USA)
Prof. A. Mendelson (*Technion*, Israel)
Prof. Y. Muraoka (*Waseda University*, Japan)
Prof. A. Nicolau (*University of California, Irvine*, USA)
Prof. G. Papadopoulos (*MIT*, USA)
Prof. Y. Patt (*University of Michigan*, USA)
Dr. S. Sakai (*ETL*, Japan)
Dr. V. Sarkar (*IBM*, USA)
Prof. H. Terada (*Osaka University*, Japan)
Prof. A. Wendelborn (*Adelaide University*, Australia)

LIST OF REFEREES

Agutter W.
Aiken A.
Albonesi D.
Alexander B.
Altman E.A
Amamiya M.
Anderson J.
Bernstein D.
Berson D.A.
Bic L.
Bodik R.
Bohm A.P.
Boku T.
Brownhill C.
Cao J.
Caton T.
Chang P.Y.
Chow J.H.
Coustet C.
Culler D.E.
Dearle A.
Degroot D.
Duesterwald E.
Dusseau A.
Ebcioglu K.
Egan G.K.
Engelhardt D.
French R.
Fuentes Jr. C.A.
Guang R. Gao
David Garza
Gaudiot J.L.
Giloi W.
Goldstein S.
Govindarajan R.
Gupta R.
Ha S.
Hains G.
Hamme J.
Hao E.
Harada K.
Hum H.
Iwata M.
Ju R.D.C.
Kawano T.
Kelly W.
Kim C.
Kim H.
Kodama Y.
Kolson D.
Konaka H.
Koren I.
Kremer U.
Kusakabe S.
Lam M.S.
Lecussan B.
Lee E.A.
Lee L.
Lemoine M.
Lim A.
Lin w.Y.
Liu L.T.
Lumetta S.
Luna S.
Maciunas K.J.
Maquelin O.
Marre D.
Martin R.
Maydan D.
Mendelson A.
Mendelson B.
Moreno J.
Muraoka Y.
Murphy B.
Murthy P.
Najjar W.
Nakamura H.
Nemawarkar S.
Nicolau A.
Ning Q.
Novack S.
O'Farrell B.
Onder S.
Onion F.
Oudshoorn M.
Palmer D.
Papadopoulos G.M.
Parks T.M.
Patt Y.
Pino J.L.
Ramanujam
Roh L.
Sakai S.
Sarkar V.
Sato M.
Schonberg E.
Seguin C.
Shankar B.
Silberman G.M.
Siron P.
Sprangle E.
Sreedhar V.C.
Srinivas M.
Sriram S.
Stark J.
Sur S.
Tackett W.
Taki K.
Tang X.
Ton S.
Terada H.
Theobald K.
Tseng C.W.
Ungerer T.
Unrau R.
Van Dongen V.
Wadge W.
Watts T.
Wendelborn A.L.
Xu Y.
Yamana H.
Yoo N.
Yoon D.K.
Zhang V.

TABLE OF CONTENTS

Part I - High-Performance Architectures

Part II - Code Generation for Multithreaded and Dataflow Architectures

Part III - Memory and Cache Issues

Part IV - Distributed Memory Machines

Part V - Multi-Level Parallelism

Part VI - Compiling for Parallel Machines

Part VII - Logic Languages

Part VIII - Application Specific Architectures

Part IX - Functional Languages, Dataflow Models and Implementation

Part X - Short Papers

PART I

HIGH-PERFORMANCE ARCHITECTURES

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

EM-C: Programming with Explicit Parallelism and Locality for the EM-4 Multiprocessor

Mitsuhisa Sato[a], Yuetsu Kodama[a], Shuichi Sakai[b] and Yoshinori Yamaguchi[a]

[a] Electrotechnical Laboratory, 1-1-4 Umezono, Tsukuba, Ibaraki 305, Japan

[b]RWC Tsukuba Research Center, Mitsui Bldg., Tsukuba, Ibaraki 305, Japan

Abstract: In this paper, we introduce the EM-C language, a parallel extension of C to express parallelism and locality for efficient parallel programming in the EM-4 multiprocessor. The EM-4 is a distributed memory multiprocessor which has a dataflow mechanism. The dataflow mechanism enables a fine-grain communication packet through the network to invoke the thread of control dynamically with very small overhead, and is extended to access remote memory in different processors. The EM-C provides the notion of a global address space with remote memory access. It also provides new language constructs to exploit medium-grain parallelism and tolerate several remote operation latencies. The EM-C compiler generates an optimal code in the presence of explicit parallelism. We also demonstrate the EM-C concepts by some examples and report performance results obtained by optimizing parallel programs with explicit parallelism and locality.

Keyword Codes: C.1.2; C.4; D.1.3; D.3.3
Keywords: Multiprocessors; Performance of Systems; Language Constructs and Features

1 Introduction

EM-C is a parallel extension of the C programming language designed for efficient parallel programming in the EM-4 multiprocessor. The EM-4[7] is a distributed memory multiprocessor which has a dataflow mechanism. Dataflow processors are designed to directly send and dispatch small fixed-size messages to and from the network. Those messages are interpreted as "dataflow tokens", which carry a word of data and a continuation to synchronize and invoke the destination thread in other processors. The dataflow mechanism allows multiple threads of control to be created and efficiently interact. In the EM-4, network messages can also be interpreted as fine-grain operations, such as remote memory reads and writes in a distributed memory environment. Remote memory access operations can be thought of as invoking a fine-grain thread to reference memory in a different processor.

EM-C provides the notion of a *global address space* which spans the local memory of all processors. The programmer can distribute data structures as single objects in the global address space. Using remote read/write messages, the language supports direct access to remote shared data in a different processor. Although it provides the illusion of global shared memory, since it does not provide a hardware shared memory coherence mechanism, frequent remote memory accesses cause large amounts of latency, resulting in poor performance. If reference patterns are static or regular, EM-C allows the programmer to convert them into coarse-grained communication in order to reference them in the local space.

Tolerating the long and unpredictable latencies of remote operations is the key to high processor utilization. Multithreading is an effective technique for hiding remote operation latency. The dataflow mechanism supports very efficient multithreaded execution. The programmer can decompose a program into medium-grain tasks to hide the remote operation latency with the computation of other tasks. We introduce the *task block* language construct to express and control parallelism of tasks.

Given the failure of automatic parallelizing compilers, many programmers want to explore writing explicit parallel programs. Some language and compiler researchers believe that explicit parallelism should be avoided, and implicitly parallel languages such as functional languages should be used. Nevertheless, we desire language constructs for expressing explicit parallelism and locality in parallel programs to achieve the best performance for the underlying hardware. Then, the ability to exploit parallelism and to manage locality of data is not limited by the compiler automatic transformation. The compiler takes care to generate optimal code in the presence of explicit parallelism. EM-C has been used extensively as a tool for EM-4 parallel programming. It may be used as a compilation target for higher level of parallel programming languages. Although the EM-C execution model is based on the EM-4 architecture, the EM-C concept is not limited to the EM-4. With software runtime systems, it can be applied to other modern distributed memory multiprocessors such as Thinking Machine's CM-5.

In Section 2, we summarize the EM-4 architecture, and introduce the EM-C language constructs in Section 3. We illustrate how these are used in EM-4 parallel programming in Section 4, and also report the performance results achieved in the EM-4 prototype. Section 5 describes the EM-C compiler and some of its optimization techniques. We compare EM-C and its execution model with related works in Section 6, and conclude with a summary of our work in Section 7.

2 The EM-4 multiprocessor

The EM-4 is a parallel hybrid dataflow/von Neumann multiprocessor. The EM-4 consists of a single chip processing element, EMC-R, which is connected via a Circular Omega Network. The EMC-R is a RISC-style processor suitable for fine-grain parallel processing. The EMC-R pipeline is designed to fuse register-based RISC execution with packet-based dataflow execution for synchronization and message handling support.

All communication is done with small fixed-sized packets. The EMC-R can generate and dispatch packets directly to and from the network, and it has hardware support to handle the queuing and scheduling of these packets.

Packets arrive from the network and are buffered in the packet queue. As a packet is read from the packet queue, a thread of computation specified by the address portion of the packet is instantiated along with the one word data. The thread then runs to its completion, and any live state in the thread may be saved to an activation frame associated with the thread. The completion of a thread causes the next packet to be automatically dequeued and interpreted from the packet queue. Thus, we call this an *explicit-switch thread*. Network packets can be interpreted as dataflow tokens. Dataflow tokens have the option of *matching* with another token on arrival — unless both tokens have arrived, the token will save itself in memory and cause the next packet to be processed.

The current compiler compiles the program into several explicit-switch threads[9]. The EM-4 recognizes two storage resources, namely, the template segments and the operand segments. The compiled codes of functions are stored in template segments. Invoking a function involves allocating an operand segment as the activation frame. The caller allocates the activation frame, depositing the argument value into the frame, and sends its continuation as a packet to invoke the callee's thread. Then it terminates, explicitly causing a context-switch. The first instruction of a thread operates on input tokens, which are loaded into two *operand registers.* The registers cannot be used to hold computation state across threads. The caller saves any live registers to the current activation frame before context-switch. The continuation sent from the caller is used for returning the result, as the return address in a conventional call. The result from the called function resumes the caller's thread by this continuation.

This calling sequence can be easily extended to call a function in a different processor remotely. When the caller forks a function as an independent thread, it is not suspended after calling the function. When using the parallel extension in EM-C, an executing thread may invoke several threads concurrently in other activation frames. Therefore, the set of activation frames form a tree rather than a stack, reflecting the dynamic calling structure.

The interpretation of packets can also be defined in software. On the arrival of a particular type of packet, the associated system-defined handler can be executed. For remote memory access, the packet handlers are provided to perform remote memory reads and writes. For a remote memory access packet, we use the packet address as a *global address* rather than specifying a thread. A global address consists of the processor number and the local address in the processor. The remote memory read generates a remote memory read packet, and terminates the thread. The remote read packet contains a continuation to return the value, as well as a global address. The result of the remote memory read packet is sent containing the value at the global address to the continuation which will receive the value. The remote memory write generates a remote memory write packet, which contains a global address and the value to be written. The thread does not terminate when the remote write message is sent.

In the EMC-R processor, incoming packets are queued and served in FIFO (first-in first-out) order. Any communication does not interrupt the execution. Threads switch explicitly at a function call and return. It is sometimes useful to form critical sections.

3 EM-C Parallel Programming Language

3.1 Global Address Space

EM-C provides a global address space and allows data in that space to be referenced through *global pointers.* A *global pointer* contains a global address, and is declared by the quantifier `global` to the pointer declaration:

```
int global *p;
```

A global pointer may be constructed by casting a local pointer to a global pointer. The same object is referenced by different PEs through the global pointer. The processor select operator `@` is used to construct a global address as follows:

`p = &x@pe_addr;` *or* `p = &x@[pe_id];`

where `pe_addr` is a processor address in the network, and `pe_id` is an unique processor identifier starting from 0. The processor select operator evaluates the left side expression within the processor given by the right side. Arithmetic on global pointers is performed on its local address while the processor address remains unchanges.

EM-C allows an array object to be distributed in a global space. A global array object is declared by the local declaration and processor mapping declaration separated by `@`.

```
data_t global A[N]@[M];
```

This corresponds to a cyclic layout of a two-dimensional `N*M` array. The data objects declared by the local declaration are laid out according to the processor mapping dimensions in a wrapped fashion starting with processor zero. If `M` is chosen equal to the number of processors, this can be viewed as a blocked layout of a one-dimensional array of `M*N` size. The element of the global array can be referenced by the processor select operator. For example, the expression `A[i]@[j]` references the i-th element of `j/`*(the number of processors)*-th `A` at the processor `j%`*(the number of processors)*.

As the default, data objects are allocated at the same local address as private object in each PE. Since all accesses to such objects are local, access can proceed without interference or delay. Any object declared without a distribution quantifier is simply replicated with one copy allocated to each PE.

3.2 Task Block for Medium Grain Parallelism

A task block is a block of code which can be executed by a thread with its own context. The *forkwith* construct is used to create a thread which executes a task block.

```
forkwith(parameters for task body){ ... task body ... }
```

The task body section contains the code executed when the thread runs. The programmer gives a list of variables to reference the values of these variables from the enclosing context. EM-C implementation allocates an activation frame for the task, and copies these values into the task's context when the task is created.

Threads may communicate and coordinate by leaving and removing data in the synchronizing data structure. EM-C provides the I-structure operation on any word in the global address space as well as the conventional lock operations. This synchronizing data structure may be used for split-phase operations to tolerate the latencies of remote operations.

The *dowith* construct blocks the parent thread until the execution of the task body terminates.

```
dowith(parameters for task body){
    ... task body ... } [ update(update variable list)]
```

The optional *update* reflects the values of specified variables into the parent's environment at termination. The task body may include any number of nested task blocks.

3.3 Data Oriented Task Block: *Where*

The *where* construct executes the task body in the PE where the data is allocated.

```
where(global pointer variable or expression) task block statement
```

This construct is useful for handling complicated data structures linked by global pointers among different PEs in an irregular application. Once a data structure has been built, related computation tasks may be allocated using the global pointer.

For convenience, the processor select operator can be used to execute a function in the specified processor. For example, the following code remotely calls `foo` at the processor specified by `pe_addr`.

```
r = foo(iargs,...)@pe_addr;
```

3.4 Constructs for Multithreading

The *parallel sections* construct, denoted by the `parallel` keyword, concurrently executes each statement in a compound statement block as a fixed set of threads. It is similar to the *cobegin/coend* or the Parallel Case statements, and is a block structured construct used to specify parallel execution of the identified sections of code. A *parallel sections* statement completes and control continues to the next statement only when all inside statements have completed.

Parallel loops differ from their sequential counterparts in that variables must be declared as private in each iteration context. Instead of a high-level parallel loop construct, EM-C provides the *iterate* construct which spawns a fixed number of the same iteration task blocks.

`iterate(`*the number of threads*`)` *task block statement* [*reduction operation*]

The optional reduction operation can be performed during the synchronization of threads for *dowith* task blocks. For example, a pre-schedule loop can be implemented as follows:

```
iterate(N_ITERATE) dowith(){
    for(s = 0, i = iterate_id(); i < N; i += N_ITERATE) s += A[i];
} reductionsum(s);
```

Here, *reductionsum* performs a sum reduction operation on variable `s`. The construct `iterate_id()` returns the identifier of the iteration task from 0 to `N_ITERATE - 1`.

The *parallel sections* and *iterate* constructs are used to tolerate the remote operation latency with other tasks by the multithreading technique.

3.5 Parallelism over Global Data Object

The *everywhere* construct parallelizes the loop over distributed data structures. It spawns the task block in every processor.

`everywhere` *task block statement* [*reduction operation*]

The following code shows an inner-product example using the *everywhere* construct.

```
data_t global A[N/N_PE]@[N_PE], global B[N/N_PE]@[N_PE];

everywhere dowith() {
   for(s = 0, i = 0; i < N/N_PE; i++)
        s += A[i]*B[i];
} reductionsum(s);
```

```
#define BLK_N (N/N_PE)
global data_t A[BLK_N][N]@[N_PE}; global data_t C[BLK_N][N]@[N_PE];
global data_t B[BLK_N][N]@[N_PE];  /* transposed */
data_t localB[N];

matrix_multiply(){
  everywhere dowith(){
    int i, j, k, l, iter; data_t t;
    j = my_id*N_BLK;
    for(iter=0 ; iter < N; iter++,j++){
        if(j >= N) j = 0;
        mem_copyout(B[j%BLK_N],(global)localB,N*sizeof(data_t))@[j/BLK_N];
        barrier();
        for (i=0;i<BLK_N;i++){
            for (t=0,k=0;k<N;k++) t += A[i][k]*localB[k];
            C[i][j] = t;
        }
    }
  }
}
```

Figure 1: Matrix Multiply (Block Copy)

In the example, the global arrays A and B without a processor select operator are referenced as local objects.

Within the context of an *everywhere* task block, the barrier synchronization can be used to synchronize every processor. To implement the barrier synchronization on EM-4, we use the packet exchange communication of a butterfly network. This allows reduction operations such as sum to be performed during packet exchange.

4 Programming in EM-C

This section describes some examples and the performance actually achieved on the EM-4 prototype. We also illustrate how to use the EM-C constructs to optimize parallel programs.

4.1 Matrix multiplication

If program communication patterns are static, they can be converted into coarse-grained block copy operation. This transformation reduces the frequency of remote operations in a distributed memory environment. In the matrix multiply with column-oriented block distribution (Fig. 1), each row in the transposed matrix B is copied into the local buffer localB in the outermost loop. The block copy mem_copyout is remotely invoked to send the block of data. To avoid remote access conflicts, the starting row is skewed according to the processor number. Processors are distinguished by the value of my_id, and the number of processors is given as a constant N_PE. Matrices A and C are referenced locally.

Figure 2 shows the performance of three versions of matrix multiply for square matrices up to size 256. Unfortunately, the EM-4 has no floating point hardware, so we measured the performance in the unit of integer operations. The lowest curve "Remote read" is for the simplest version which accesses matrix B in the innermost loop by remote memory read. This version only uses a single thread. Although the performance can be improved by multithreading technique, it cannot exceed the block version due to the overhead of fine-grain remote reads in this program. The curve labeled "Block Copy" is for the version described above. The "Shift" version uses the cyclic shift algorithm, which is usually used

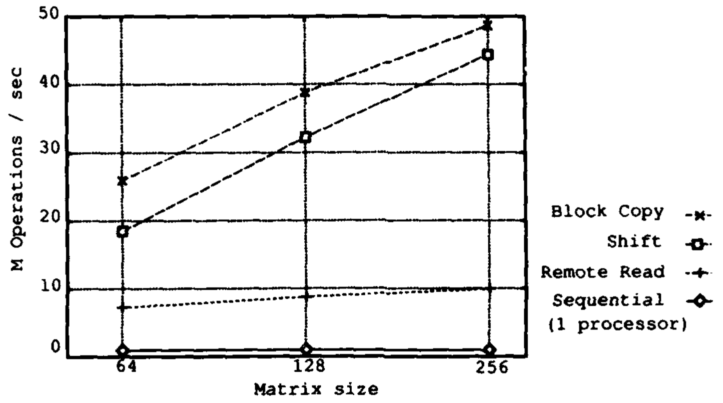


Figure 2: Performance of matrix multiply (64 processors)

in conventional message passing model. In EM-C, the shift operation has the extra cost to move data from message buffers to the program data structure.

4.2 Knapsack Problem

Given n objects with weights W_i and profits P_i, and a knapsack capacity M, the knapsack problem is to determine the subset of objects which maximumize total profits. The knapsack problem gives rise to a search space of 2^n of objects, which can be depicted as a binary tree, where the root represents an empty knapsack, and going down to $level_i$ subtree represents the choice of $object_i$.

We implemented *the Branch and Bound* algorithm presented in [1], as shown in Figure 3. Given a partial solution, a lower bound for the best total solution can be computed by adding objects with maximal profit weight ratio until an object exceeds the knapsack capacity, while an upper bound can be computed by adding the part of the objects that fill the capacity. Subtrees are cut by estimating the upper bound of a partial solution and comparing it to a *shared variable GLow* containing the current best lower bound in all the processors.

The array for weights and profits are replicated in each processor, and are referenced locally. The variables `next1_pe` and `next2_pe` contain the neighbor processor addresses in the EM-4's Omega network topology. Remote function calls for the subtrees are done within parallel sections, and parallelism spreads in the form of tree in omega topology. To avoid the exhaustion of resources, parallel remote calls are switched into sequential local calls when the number of activations, counted by the variable `thread_count`, exceeds the limit `THREAD_LIMIT` (50 in the current implementation). Since a thread is executed exclusively, this variable is updated in a critical section. Figure 4 shows the speedups of 30 random data from 20 objects to 60 objects with 80 processors. The trend seems to be that the harder the problem becomes, the better performance the parallel program obtains. For small problems, the speculative execution does not work effectively, and may sometimes make them even slow. To manage the shared variable GLow, we implemented two versions of `Fetch_GLow` and `Update_GLow`. The version "remote read/remote update" always references and updates the GLow in processor 0. In "local read/broadcast update",

```
int W[N], P[N];  /* weight and property */
global int GLow; /* shared variable */

int knapbb(i,cp,M) int i,cp,M;
{  int lwb,upb,Opt, m,ii,curlow, l,r;
    static int thread_count = 0;

    Opt = cp;
    if (i < N && M > 0) {        /* compute lower and upper bounds */
        for(lwb=cp,ii = i,m = M; (ii < N && m >= W[ii]); ii++){
            m -= W[ii];  lwb += P[ii];    }
        if(ii < N) upb = lwb + (m*P[ii])/W[ii]; else upb = lwb;
        curlow = Fetch_GLow();    /* fetch global variable */
        if(curlow < lwb) Update_GLow(lwb);
        else if(upb < curlow) return(Opt);
        if (M >= W[i]){
            if(thread_count < THREAD_LIMIT){
                thread_count++;
                parallel {    /* parallel call */
                    l = knapbb(i+1,cp+P[i],M-W[i])@next1_pe;
                    r = knapbb(i+1,cp,M)@next2_pe;
                }
                thread_count--;
            } else {          /* sequential call */
                l = knapbb(i+1,cp+P[i],M-W[i],0);
                r = knapbb(i+1,cp,M,0);
            }
            Opt = MAX(l,r);
        }  else  Opt = knapbb(i+1,cp,M,0);
    }
    return(Opt);
}
```

Figure 3: Knapsack problem in EM-C

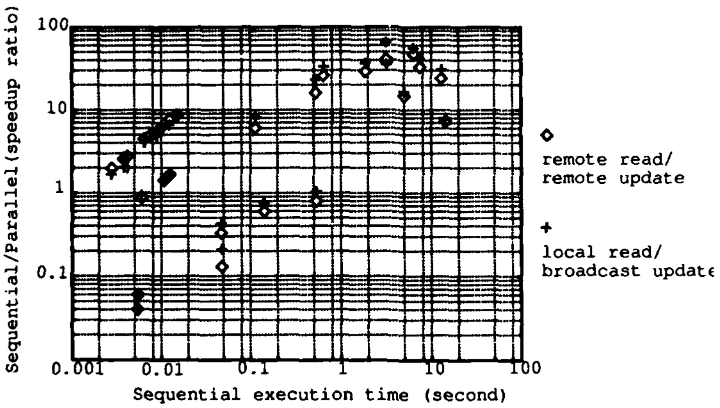


Figure 4: Performance of knapsack problem (80 processors)

GLow is replicated in every processor and referenced locally. The value of GLow is updated by invoking a thread to broadcast the new value. This policy is better in the harder problems, because reference occurs more frequently than update.

```
Read wire data, and distribute them for all PEs.
for (repeat 2 times){
    everywhere() dowith(){
        /* this task block is executed in every PE */
        iterate(#threads for wiring) dowith(){
            pick a wire in the current PE.
            rip up the previous route from the cost array.(if not first time)
            make pin pairs by building minimum spanning tree
            for(each generated pin pairs){
                iterate(#threads for routing)
                    dowith(){
                        pick one possible route.
                        calculate the route cost with the current cost array.
                    }
                choose the best route with the minimum cost.
                }
            record the best routes, and update the cost array.
        }
        /* for load balancing */
        if a wire in other PEs is not routed yet, move the wires and route it.
    }
}
```

Figure 5: Outline of Locus Route in EM-C

4.3 LocusRoute

LocusRoute[6] is a standard-cell routing program. The cost array is a main data structure in LocusRoute, which is referenced and updated by wiring processes. In the EM-C implementation, the cost array is statically distributed in a global address space. Wire data is distributed in round-robin fashion, and is referenced locally in wiring processes.

The outline of the EM-C implementation is illustrated in Figure 5. The major modification from the original program is eliminating several global variables to expose parallel loops. Parallel loops are executed inside *everywhere* sections. Since the cost array is distributed in each processor, its elements are accessed by remote memory operations. To tolerate the latency of remote reference, multiple wires are routed in parallel even in the same processor by the *iterate* construct. To find the best routes, a few threads are also generated dynamically by *iterate*. At end of the loop, each routing process tries to move unrouted wires in other processors for load balancing. With 64 processors, we achieved a speedup of 31.5 for the input circuit (`Primary2`) which has 3817 wires in an 1920 grid by 20 channel area. Through careful experimentation, we found that four to eight threads in each processor is sufficient to hide the remote operation latency in the EM-4 prototype. Since aggressive multithreading increases pressure on the network, it can have a negative impact on performance as the number of processors increases. Detailed results are reported in [8].

5 The EM-C Compiler

5.1 Intermediate Code

The compiler takes an expression tree generated by the front-end, and expands it into the intermediate code. In the intermediate code of the EM-C compiler, we introduce the following special operations to represent parallel programs.

LFORK *label,output-dir,break-or-continue,optional-operands, ...*

LFORK invokes a new thread running from the specified label in the current activation frame. The operand *break-or-continue* indicates whether the current thread terminates after this operation.

GFORK *label,output-dir,break-or-continue, optional-operands, ...*

This operation indicates an indirect control flow. GFORK sends the continuations for the specified label to invoke a new thread in an external activation frame. The thread is resumed by the return value sent from the external thread. The return value is obtained at the destination of a GFORK operation by the INPUT(*input-dir*) operation.

SEND *data,destination,optional-operands, ...*

Sends the *data* to the *destination*. This operation is used for remote asynchronous write and forking independent threads.

The optional operands may specify the value carried across threads. The *output-dir* and *input-dir* operands identify input and output for synchronization. The value may be either `lop`, `left`, or `right`; `lop` invokes the destination thread immediately, and `left` and `right` are used to join two threads. This synchronization is done by dataflow matching.

For example, the remote read is compiled into the following code:

```
      GFORK C0,lop,break,"REMOTE_READ", global address
C0: variables = INPUT(lop)
```

The INPUT operations are removed by assigning the operand registers to the destination variables in register allocation phase.

Every function call is converted into GFORK operations in the intermediate code, because each function is compiled as an individual thread.

In the intermeditate code, the task block code is surrounded by the TASK_BEGIN and TASK_END code annotated with the type of control. These codes are directly translated into the corresponding runtime assembly code in the code generation phase.

5.2 Extended Control Flow Graph

The EM-C compiler constructs the Control Flow Graph (CFG) as an intermidate data structure. Our CFG is extended to introduce special edges associated with LFORK and GFORK. In CFG, the following transformations are performed to optimize the program:

- *Re-partitioning.* — Maximizing the thread run length reduces the frequency of saving and restoring context, and increases locality. We use the Dependence Set algorithms[4] to recognize possible partitioning. The algorithm is modified for the extended CFG because the program is represented by a control flow graph rather than a data flow graph. When the nodes connected by a LFORK edge are in the same dependence set of GFORK edges, these nodes can be executed in the same thread.

- *Null thread elimination.* — A null thread[1] is defined as a basic block which contains no effective instructions except an LFORK operation to trigger other threads. For example, suppose the fork operation in node A triggers the null thread of node B, and B triggers C, then B can be eliminated by replacing the fork operation A as to directly trigger C.

[1] This is the same as a Null Sequential Quatum in [4]

5.3 Register Allocation

We use the register coloring algorithm for register allocation. In our execution model, the live ranges of variables in concurrently executed nodes do not interface with each other. Different spill locations must be assigned to these variables, however, because their order of execution is unknown at compile time, and the live values of the variables are saved at any thread boundary.

At the beginning of each thread, the destinations of INPUT are assigned to the operand registers. Peep hole optimization may combine an instruction into dataflow matching if possible. When dataflow matching is used only for synchronizing controls, a single live value may be carried with the synchronization packet.

Code scheduling across basic blocks is not implemented in the current compiler.

6 Related Work

Eicken et. al. [10] presented *Active Messages* as an asynchronous communication mechanism which directly invokes a user-level instruction sequence in the message. The performance of Active Messages on CM-5 indicates that it takes a few tens of machine cycles to create a thread in a different processor using hardware directly from the user program. They proposed an extension of the C programming language, called Split-C[2], which provides split-phase remote memory operations using Active Messages and can overlap asynchronous communication with computation. Split-C also provides a global address space. The major difference is that Split-C provides a set of operations for split-phase assignment to allows computation and communication to be overlapped, rather than providing language constructs for multithreading as in the EM-C.

Recent works in dataflow research have adopted the *explicit-switch* thread. Culler[3] proposed a software explicit-switch thread model, called a threaded abstract machine (TAM) using Active Messages, which is similar to the EM-4 architecture. The TAM was designed to compile the functional programming language Id90 into it. Although our EM-C is designed for the EM-4 architecture, we believe that the basic concept of EM-C can be applied to conventional high performance multiprocessors with a similar approach.

Jade [5] is a data-oriented language for parallelizing programs written in a serial, imperative language. It supports coarse-grain tasks which are augmented with high-level data usage information. Its compiler and runtime systems use this information to synchronize tasks. EM-C task blocks can be used for the same style of parallel programming.

7 Summary

In this paper, we introduced EM-C, a parallel extension of C designed for efficient parallel programming in the EM-4 multiprocessor. The EM-4 has a dataflow mechanism, which is extended to fine-grain remote memory access and is able to support very efficient multithreading. EM-C allows programmers to exploit and control parallelism, and to enhance locality for optimizing parallel programs. It provides the notion of a global address space for remote data references, and simple data distribution in that space. EM-C's task block and parallel sections construct are used for multithreading to exploit medium-grain parallelism and to hide remote memory access latencies. The *where* and *everywhere* task block constructs allow data-oriented threads of task blocks to execute depending on the data distribution in the global address space. We illustrated these language concepts with various sample programs. T^he EM-C programs are compiled into explicit-switch threads to be invoked and synchroni..d with the dataflow matching mechanism. The compiler optimization eliminates redundant synchronizations and communications to generate optimal code in the presence of explicit parallelism.

Acknowledgements

We wish to thank Dr. Ohta, Director of the Computer Science Division of ETL, for supporting this research, and also the staff of our Computer Architecture Section for their fruitful discussion. We thank Prof. Böhm for valuable discussion on the implementation of knapsack problem.

References

[1] A.P.W. Böhm and Greg Egan. Five ways to fill a knapsack: Implementation of the Knapsack Problem in SISAL. In *Proc. of 2nd Sisal Users Conf.*, pages 9–20, 1992.

[2] D. E. Culler, A. Dusseau, S. G. Goldstein, S. Lumetta T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. of Supercomputing '93*, Nov. 1993.

[3] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proc. ^f 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[4] R. A. Iannucci. *Parallel Machines: Parallel Machine Languages.* Kluwer Academic Publishers, 1990.

[5] M. S. Lam and M. C. Rinard. Coarse-Grain Parallel Programming in Jade. In *Proceedings of the 3rd SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 94–105, April 1991.

[6] J. Rose. The Parallel Decomposition and Implmentation of an Itegrated Circuit Global Router. In *Proc. of PPEARS 88*, pages 138–145, 1988.

[7] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proc. of the 16th Annual International Symposium on Computer Architecture*, pages 46–53, June 1989.

[8] M. Sato, Y. Kodama, S. Sakai, and Y. Yamaguchi. Experience with Executing Shared Memory Programs using Fine-Grain Communication and Multithreading in EM-4. In *Proc. of International Parallel Processing Symp. '94*, pages 630–636, April 1994.

[9] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumuara. Thread-based Programming for the EM-4 Hybrid Dataflow Machine. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, pages 146–155, May 1992.

[10] T. von Eicken, D.E. Culler, S.C.Goldstein, and K.E.Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

A Fine-Grain Threaded Abstract Machine

Jesper Vasell

Department of Computer Engineering, Chalmers University of Technology,
S-412 96 Göteborg, Sweden

Abstract: This paper presents an execution model for exploitation of fine-grain parallelism. It is described by means of an abstract machine, S–TAM, that offers flexibility in the trade-off between static and dynamic methods for scheduling and allocation It is based on multithreading as a means of providing flexible, compiler-controlled scheduling. The allocation of processing resources is both static, requiring compile-time allocation of a processor for each thread, and dynamic in the form of multiple, dynamically allocated functional units in a processor.

Experimental results show that a system using mixed static and dynamic allocation gives a performance which is comparable to that of a system based entirely on dynamic allocation.

Keyword Codes: C.1.2; D.1.3
Keywords: Multiprocessors; Concurrent Programming

1 Introduction

Today the most common type of architecture for exploitation of very fine-grain parallelism is the *superscalar* processor. Several researchers have studied the limitations of these architectures ([9, 4]) and proposed variations based, for instance, on *speculative execution* as a means for achieving a higher degree of parallelism. It seems, however, as though the degree of parallelism that can be found and exploited by these architectures is limited by their inherently sequential execution model.

Another type of fine-grain parallel architecture is the *VLIW* (Very Long Instruction Word) architecture. In contrast to a superscalar architecture, a VLIW architecture makes use of static scheduling and allocation of processing resources, to exploit parallelism. Since superscalar architectures are based on *dynamic* scheduling and allocation, the primary difference between VLIW and superscalar execution lies in how and when scheduling and allocation of processing resources are performed.

The chief disadvantage of dynamic scheduling, compared with static scheduling, is the overhead in both time and hardware that is incurred by the scheduling mechanism. Furthermore, dynamic scheduling often becomes local in the sense that it only considers a small portion of a program at a time. On the other hand, dynamic scheduling is more general than static scheduling, which is best suited for regular computations with a statically determined control flow. For irregular computations, it is often difficult to perform efficient static scheduling and allocation.

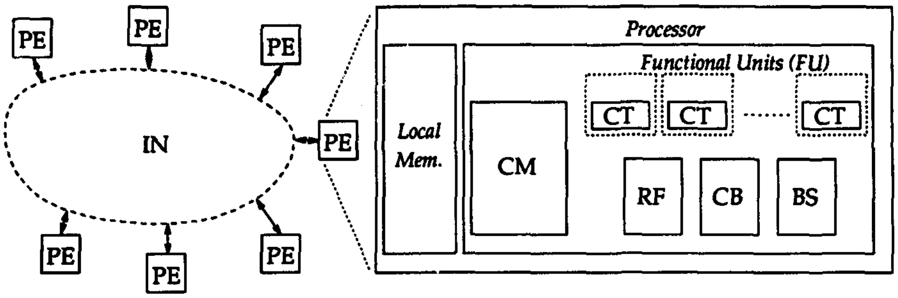


Figure 1: An S–TAM processor.

This paper presents an execution model for very fine-grain parallelism that constitutes a new and flexible trade-off with respect to static and dynamic methods for scheduling and allocation. The execution model is described by means of an abstract machine called S–TAM (Static Threaded Abstract Machine). Its name is chosen to reflect that some of the underlying ideas are inspired by the TAM abstract machine [1]. For instance, one important goal of S–TAM is to make scheduling and allocation *explicit* and visible to the compiler.

The flexibility in scheduling offered by S–TAM comes primarily from the fact that it is a multithreaded execution model, in which threads are dynamically scheduled in a data-driven fashion, whereas the instructions constituting a thread are statically scheduled. Threads are formed so as to avoid parallelism within threads, i.e. parallelism should be expressed as parallelism between threads.

The flexibility in resource allocation comes from the fact that S–TAM is organized as a number of statically allocated processors, consisting of dynamically allocated functional units. The basis for the allocation is thus static, as each processor is assigned one or several threads at compile-time. However, this allocation is not as critical as for a VLIW architecture, and can be performed with a very simple allocation algorithm, owing to the fact that each processor has several functional units to handle parallelism between threads on the same processor.

2 S–TAM

A system based on S–TAM consists of a number of processing elements (PE), connected via some interconnection network (see Figure 1). Each PE consists of some local memory and an S–TAM processor with the following storage resources:

CM *Code Memory*
Holds threads of instructions and synchronization points.

RF *Register file*
A set of general purpose registers R0–Rn.

CB *Continuation Buffer*
A buffer holding thread continuation points in CM. A continuation point indicates an

instruction or synchronization point from which execution of a thread may proceed.

BS *Blocking Store*
A memory holding blocked points in CM. A blocked point indicates an instruction or synchronization point where execution of a thread has been blocked.

CT *Current thread*
Pointer to the currently executed instruction in CM. Functions as an ordinary program counter.

A (CM,RF,CB,BS,CT) collection denotes an S-TAM *processor* with a single point of execution. The registers in a register file can be accessed only by instructions stored in the corresponding code memory, and are allocated globally for all threads in the code memory, thereby allowing these threads to communicate via registers rather than messages.

Parallelism is introduced by the presence of multiple S-TAM processors, and by multiple *functional units* in a processor. In Figure 1 this is represented by multiple CT:s served by a single CB and BS. The difference between the two forms of parallelism is that threads are *statically* allocated to processors, but *dynamically* allocated to functional units. This means that a thread may start executing on one functional unit until it becomes suspended, and later resume execution on a *different* functional unit.

An S-TAM thread is a sequence of instructions from an instruction set, consisting primarily of conventional instructions operating on register operands. There are also instructions for memory operations, performed as split-phase transactions.

Communication between threads, and the associated synchronization, is embodied in a SEND instruction. A SEND instruction and a *synchronization point*, represent a communication path between two threads:

```
Th1:      :             Th2:    :
          :                     :
SEND R1,Th2.S1  ---------->  ADD S1,R2,R3
          :                     :
          :                     :
```

The SEND instruction creates and sends a message to the synchronization point (denoted S1). When the message is received, the value is stored in the synchronization point, and an acknowledgement message to the SEND instruction is generated as soon as the receiving thread has read the value from the synchronization point.

For synchronization purposes, there is a state associated with each SEND instruction and synchronization point. A synchronization point can be in any of the states BLOCKED, EMPTY or FULL, whereas an instruction can be either BLOCKED, UNACK (UNACKnowledged) or ACK (ACKnowledged). A thread is executed sequentially using the Current Thread pointer (CT) as a program counter. However, execution of a thread cannot proceed beyond an EMPTY synchronization point or an UNACKnowledged SEND instruction. In these cases, the currently executing thread is suspended and placed in the Blocking Store (BS), and execution continues with some other thread from the continuation buffer (CB). A thread is moved from BS to CB as soon as the processor receives a message relating to the SEND instruction or synchronization point on which the thread was suspended.

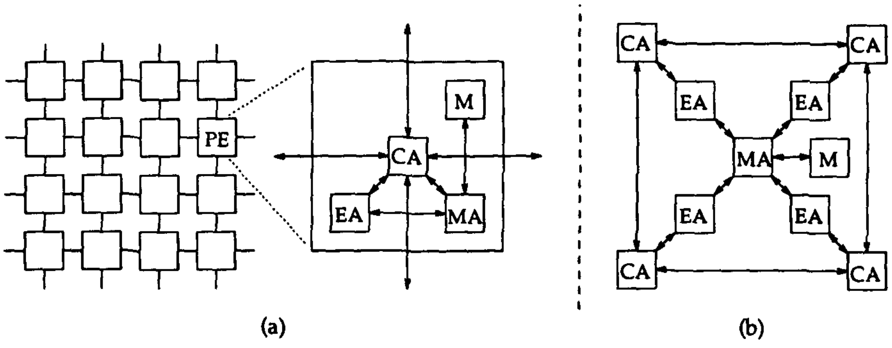


Figure 2: Organization of an S–TAM system.

3 Organization of an S–TAM System

Figure 2(a) depicts a possible organization of an architecture based on the S–TAM execution model. This organization is similar to that of other parallel architectures, such as iWarp [6], J-Machine [2] and *T [5]. The main difference lies in the granularity. As this is meant to be a very *fine-grained* parallel architecture, the communication network and the processing elements must be able to handle small messages with a low latency.

Each processing element (PE) has the following parts:

Execution Agent (**EA**): Implements the S–TAM execution model.

Memory Agent (**MA**): Serves as an interface to the memory system.

Communication Agent (**CA**): Performs message handling and routing.

Memory (**M**): Local memory associated with each processing element.

Memory is physically distributed in order to achieve good scalability. Whether the logical organization of the memory should be a global shared address space or distributed local address spaces is not directly implied by the organization of the architecture. Although this is an important aspect, this question is not considered here. The reason is that neither the organization nor the execution model represent a preference for one or the other.

The organization shown in Figure 2(b) has been used in the simulations described in Section 5. It consists of 4 processors, sharing a single memory and memory agent. It is expected that this organization could be realized as a single VLSI circuit, using a modern CMOS technology.

4 Compiling for S–TAM

The process of compiling for a multithreaded execution model basically means transforming a computation into a set of threads, each of which represents a separately schedulable subcomputation. There are four main steps in compiling for S–TAM: *Data flow graph* *generation*, *Clustering*, *Code generation*, and *Processor allocation*. For the experiments reported in Section 5, a compiler for *semi-static* data flow [8] has been used. Even though the instruction set of S-TAM is tailored for this particular form of DFGs, it is general enough to allow code generation from other forms of DFGs as well.

4.1 Clustering and Code Generation

The purpose of a clustering algorithm is to create clusters of fine-grain operations that can be scheduled as sequential threads of instructions. Furthermore, the clustering should minimize the synchronization overhead. For S-TAM it is important to use a clustering algorithm which attempts to retain all of the fine-grain parallelism, as the execution of a thread is entirely sequential. This is more important than creating long threads with a minimum of synchronization overhead, even though these aspects are of interest as well.

When forming clusters for S-TAM, locality is of great importance, both because exploiting locality has the effect of reducing the number of synchronizations associated with a thread and because locality implies dependency between operations, which prohibits parallel execution. If operation A depends directly on the result of operation B, then, by placing these operations in the same cluster, this locality in communication can be taken advantage of, and it is guaranteed at the same time that, as a result of this dependency, no parallelism is lost.

The purpose of the code generation phase is to create a sequence of S-TAM instructions for each cluster. Because the clusters are sequential, this is basically a matter of generating the necessary instructions for each DFG node in the cluster in the order they were added to the cluster. In most cases, a cluster gives rise to a single thread. There are, however, exceptions from this in conjunction with nodes which are non-strict in their inputs.

See [7] for a more comprehensive description of the clustering and code generation.

4.2 Processor and Storage Allocation

The last step in compilation for S-TAM consists of allocation of processing and storage resources. Allocation of processing resources means allocating an S-TAM processor for each thread, i.e. for a multiprocessor S-TAM, to decide which CM should hold a specific thread. The significant aspect of this is that it is *statically* decided which processor should execute a thread.

The storage allocation, which is performed *after* the processor allocation, concerns the allocation of physical storage such as registers and synchronization points. This is performed individually for each thread, under the assumption that each thread needs a distinct set of registers. The storage allocation issues will not be discussed any further. For the experiments and measurements presented in Section 5, a compiler and simulator were used, which assume that a sufficient number of registers and synchronization points are available to make it possible to allocate a distinct set for each thread.

The following describes a simple processor allocation strategy, based on knowing which subcomputations (threads) may *not* be executed in parallel. It is guided by the data dependencies between subcomputations. If there is a dependency, such that one subcomputation depends on the result of another subcomputation, parallel execution is, in principle, not possible. However, since a thread can communicate with other threads at any point during its execution, two threads may be *partly* parallel. To handle this situation, the processor allocation strategy used for S-TAM is based on the notion of *overlap* between threads, which serves as a measure of the *potential* for parallel execution of the threads.

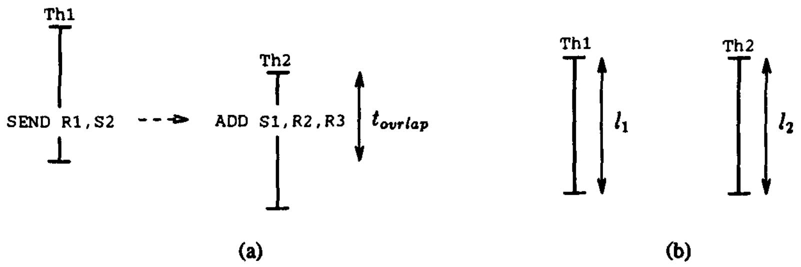


Figure 3: Measuring *overlap* between threads.

In practice, there are two distinct cases for which the *overlap* between a pair of threads must be defined. Figure 3(a) shows the case in which there exists a dependency between two threads. In this case, the overlap between the threads is marked t_{ovrlap}, and is measured as the number of instructions, before and after the synchronization, that can execute in parallel. Figure 3(b) shows the case in which there is no dependency between the threads. Deciding the *potential* for parallel execution in this case is obviously difficult, but a conservative approach is used and the overlap is taken to be the length of the shortest thread, i.e. $min(l_1, l_2)$.

The goal of the processor allocation algorithm is to minimize the sum of the overlaps between all threads allocated to a processor. It does this by adding threads one by one to the processors, selecting the processor that gives the smallest total overlap. However, given that communication between threads is associated with a cost which is likely to increase with the distance, locality is desirable, i.e. even though there is a potential for parallelism between two threads, it may be more efficient to execute both at the same processor. This corresponds to accepting a certain amount of overlap between threads on the same processor. This is represented by a *threshold* in the definition of the overlap between two threads:

$$\left\{ \begin{array}{ll} overlap(t_1,t_2) = \left\{ \begin{array}{ll} t_{ovrlap}, & \text{if } t_{ovrlap} > t_{thresh} \\ 0, & \text{otherwise} \end{array} \right. & \text{(a)} \\ overlap(t_1,t_2) = min(l_1,l_2) & \text{(b)} \end{array} \right.$$

The overlap is thus set to 0 unless t_{ovrlap} exceeds t_{thresh}. The following is an algorithm for allocation of processors based on this notion of overlap:

```
FOR each processor $P_n$
    randomly choose a thread, and assign it to $P_n$
END
FOR each remaining thread $t$
    let $proc\_ovrlap(t,n) = \sum_{t_k \in \text{threads on } P_n} overlap(t,t_k)$
    assign $t$ to the processor $P_n$ with the smallest $proc\_ovrlap(t,n)$
END
```

This algorithm uses the overlap to select a processor on which a thread t should be executed. For each processor P_n, *overlap*(t, t_k) is computed for each thread t_k already allocated to it. The sum of these overlaps, *proc_overlap*(t, n), is used as a measure of the overlap between thread t and the processor P_n. Thread t is then assigned to the processor for which *proc_overlap*(t, n) is smallest.

5 Experimental Results

This section presents and discusses some experimental results that serve as an evaluation of S–TAM. It is important to note that the results primarily concern the *execution model*, and do not relate to a specific implementation. However, in addition to being an evaluation of the execution model, the results also give important information concerning implementation of S–TAM.

The experiments have been performed using the compilation strategy described in the previous section. Two simulators have been used: an *ideal* simulator, and a simulator that models the organization shown in Figure 2(b).

The ideal simulator has been used as a reference, and all execution times have been normalized with respect to this simulator which has infinitely many, dynamically scheduled functional units. There are no overhead or latency associated with communication and synchronization, and all instructions execute in one cycle.

The second simulator models a more realistic S–TAM system, with the following characteristics:

- *Deterministic routing between processors.* All messages between two processors are routed along the same communication links.
- *One-cycle latency for EA-CA communication.* Transferring a message generated by the execution agent (EA) to the communication agent requires one cycle.
- *One-cycle latency for CA-CA communication.* Transferring messages between two communication agents requires one cycle.
- *Infinitely many registers and synchronization points.* No actual allocation of registers or synchronization points is performed.
- *One-cycle latency for local messages.* Messages between threads on the same processor have a latency of one cycle.
- *One suspension without penalty per cycle.* If a processor has multiple functional units, one of these may suspend execution of a thread and continue the execution of another thread within a single cycle.
- *Single-cycle instruction execution.* All instructions require one cycle to execute.
- *One memory request per cycle.* The memory system accepts only a single memory request per cycle.

As the simulator is parameterized with respect to the number of functional units, it is possible to simulate a range of different configurations. The number of processors can also be varied from 1 to 4 simply by performing the processor allocation for the desired number of processors. In the following, the configurations will be named **P**x**F**y, where x

Program	P1	P2	P3	P4	P2F2			P1F4		
	F1	F1	F1	F1	100%	95%	90%	100%	95%	90%
Isort	1.72	1.38	1.39	1.47	1.14	1.50	1.81	1.00	1.42	1.69
Convolution	2.11	1.63	1.50	1.52	1.23	1.74	2.19	1.06	1.69	2.11
Matmult	2.93	1.90	1.56	1.47	1.30	1.69	1.95	1.17	1.54	1.79
Matmult9	8.38	4.46	3.15	2.71	2.45	2.95	3.14	2.51	3.02	3.27
Find	3.17	2.17	1.86	1.75	1.47	1.95	2.31	1.34	1.93	2.23
Simple	8.09	4.23	3.06	2.46	2.35	3.03	3.19	2.34	3.00	3.36

Table 1: Relative Performance.

is the number of processors (1 ... 4) and y is the number of functional units per processor (1 ... 4). For example, **P2F4** denotes a configuration with 2 processors and 4 functional units per processor.

5.1 Benchmarks

The following benchmarks have been used:

`isort l`: Insertion sort applied to the list `[20...1]`.

`convolution l1 l2`: Convolution of two vectors, represented as lists. Applied to the lists `[1..5]` and `[1..15]`.

`matmult m1 m2`: Multiplication of $n \times n$ matrices. Applied to matrices of size 9×9.

`matmult9 m1 m2`: Multiplication of 9×9 matrices. Derived from the `matmult` benchmark by unfolding the inner loop.

`find p s`: String search program. Searches for pattern `p` in string `s`. `"230501"` is used as pattern searched for in string `"12123454512316789"`. The character `'0'` in the pattern functions as a wildcard character, matching any sequence of characters of at least length 1.

`simple`: Hydrodynamics simulation of the flow velocity for a fluid in a cross-section of a sphere.

5.2 Results

Table 1 shows the relative performance of a number of S–TAM configurations with varying number of processors and functional units. For **P2F2** and **P1F4** three different numbers are given that corresponds to different memory system behavior. The execution time has been measured for a memory system with hit-rates of 100%, 95% and 90%. A cache hit is assumed to have a latency of 1 cycle, and cache misses have a latency which is randomly distributed between 20 and 100 cycles.

The first four columns show the performance of statically allocated configurations, i.e. configurations with varying number of processors but only a single functional unit per processor. A comparison of these configurations indicates how well static allocation scales when the number of processors is increased.

The `find` and `matmult` benchmarks scale reasonably well up to 3 processors, whereas the two smallest ones, `isort` and `convolution`, scale poorly and, in fact, reach a point at which negative speed-up is obtained. The situation is better for `matmult9` and `simple`, which is explained by the higher degree of parallelism (8–9 with ideal execution) in these programs.

It can be seen from the results presented here that good scalability under static allocation requires that the parallelism of the computation is larger than the number of processors in the architecture. Efficient exploitation of lower degrees of parallelism requires either more elaborate allocation methods or the dynamic allocation used in configurations with multiple functional units per processor.

A comparison of the **P4F1**, **P2F2** and **P1F4** configurations, which represent different trade-offs between static and dynamic allocation, shows that execution time for **P4F1** is 25-45% higher than **P1F4** for the benchmarks that have the lowest parallelism. For `matmult9` and `simple`, the difference is smaller. However the difference between **P1F4** and **P2F2** is generally smaller than the difference between **P1F4** and **P4F1**, showing that it is not necessary to resort to completely dynamic allocation to get good performance. In fact, for the larger benchmarks **P2F2** gives as good, or even better, performance than **P1F4**.

An important aspect of a multithreaded execution model is its potential for hiding long latencies in communication between processors and memory. Table 1 shows the effect that varying memory system latencies has on the execution time for **P2F2** and **P1F4**. For 95% hit-rate, the relative increase in execution time ranges from 20%, for the benchmarks with the highest degree of parallelism, up to 58%. The corresponding numbers for 90% hit-rate are 30% up to 100%. The variation between configurations with the same total number of functional units is small, indicating that the capability to tolerate these latencies is relatively independent of whether static or dynamic allocation is used.

6 Related Work

Many aspects of S–TAM are similar to an execution model called *Processor Coupling*, proposed by Keckler and Dally [3]. Processor coupling attempts to combine static and dynamic scheduling to efficiently exploit fine-grain parallelism in a multithreaded execution model. In processor coupling, a thread can be viewed as a sequence of VLIW instructions. The processing resources for an instruction are statically allocated among a number of functional units organized into clusters. Such a cluster is comparable to an S–TAM processor with multiple functional units.

Keckler and Dally reports a number of experimental results from simulations of a processor implementing processor coupling. Regarding such as issues as functional unit utilization and memory system latency, the numbers are similar to the corresponding numbers for S–TAM.

7 Conclusions

S–TAM is an abstract machine that offers a flexible trade-off between static and dynamic methods for scheduling and allocation. It is based on multithreading as a means of providing flexibility in scheduling. The dynamic scheduling is based on mechanisms that can be implemented with relatively simple hardware. Flexibility in allocation of processing resources is achieved by multiple, statically allocated, *processors* consisting of multiple, dynamically allocated, *functional units*.

Experimental results show that a system using mixed static and dynamic allocation in this way gives a performance which is comparable to that of a system based entirely on dynamic allocation. Furthermore, these results are obtained using a very simple algorithm for processor allocation, which does not rely on any global analysis of programs.

Although the overall results are encouraging, a more detailed analysis of an actual implementation is needed in order to give definite results regarding the usefulness of the S-TAM approach. This would make a detailed comparison between S-TAM, superscalar and VLIW architectures possible. For instance, it is not yet clear whether the cost of an S-TAM implementation is comparable to a superscalar processor with the same number of functional units.

References

[1] D.E. Culler, A. Sah, K.E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *4th Int. Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, April 1991.

[2] W.J. Dally, J.A.S. Fiske, J.S. Keen, R.A. Lethin, M.D. Noakes, P.R. Nuth, R.E. Davison, and G.A. Fyler. The Message Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro*, 12(2):23–39, April 1992.

[3] S.W. Keckler and W.J. Dally. Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism. In *The 19th Annual Int. Symp. on Computer Architecture*, pages 202–213, Gold Coast, Australia, May 1992. ACM Press.

[4] Monica S. Lam and Robert P. Wilson. Limits of Control Flow on Parallelism. In *The 19th Annual Int. Symp. on Computer Architecture*, pages 46–57, Gold Coast, Australia, May 1992. ACM Press.

[5] R.S. Nikhil, G.M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *19th Annual Int. Symp. on Computer Architecture*, pages 156–167. ACM Press, May 1992.

[6] C. Peterson, J. Sutton, and P. Wiley. iWarp: A 100-MOPS, LIW Microprocessor for Multicomputers. *IEEE Micro*, 11(3):22–25, 88–94, June 1991.

[7] Jesper Vasell. *Architectures and Compilation Techniques for a Data-Driven Processor Array*. PhD thesis, Dept. of Computer Engineering, Chalmers University of Technology, S-412 96 Göteborg, Sweden, March 1994.

[8] Jonas Vasell. Static vs. Dynamic Strategies for Fine-Grain dataflow Synchronization. In G.R. Gao M. Cosnard and G.M. Silberman, editors, *Proceedings of the IFIP WG 10.3 Int. Conf. on Parallel Architectures and Compilation Techniques*. Elsevier Science B.V., August 1994.

[9] David W. Wall. Limits of Instruction-Level Parallelism. In *Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, Santa Clara, California, April 1991. ACM Press.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Tradeoffs in the Design of Single Chip Multiprocessors

David H. Albonesi and Israel Koren

Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003, USA

Abstract: By the end of the decade, as VLSI integration levels continue to increase, building a multiprocessor system on a single chip will become feasible. In this paper, we propose to analyze the tradeoffs involved in designing such a chip, and specifically address whether to allocate available chip area to larger caches or to large numbers of processors. Using the dimensions of the Alpha 21064 microprocessor as a basis, we determine several candidate configurations which vary in cache size and number of processors, and evaluate them in terms of both processing power and cycle time. We then investigate fine tuning the architecture in order to further improve performance, by trading off the number of processors for a larger TLB size. Our results show that for a coarse-grain execution environment, adding processors at the expense of cache size improves performance up to a point. We then show that increasing TLB size at the expense of the number of processors can further improve performance.

Keyword Codes: C.1.2; C.4; C.5.4
Keywords: Multiprocessors; Performance of Systems; VLSI Systems

1 Introduction

VLSI integration levels continue to rapidly increase, so much so that it has been predicted that by the end of the decade, a one inch square chip with 0.25 micron technology will be available[17]. At this level of integration, it will become possible to build a multiprocessor system with several of today's microprocessors on a single chip. Microprocessor designers will have a wide design space to explore, and many tradeoffs to make between processor architecture, cache hierarchy and TLB organization, interconnect strategies, and incorporating multiple processors on the chip. In this paper, we begin to address the design of such *high integration microprocessor architectures*. In particular, we evaluate performance tradeoffs in allocating chip resources to larger caches versus more processors. We also investigate how elements of the processor architecture (in particular TLB size) affect design decisions.

The rest of this paper is organized as follows. First we discuss the multiprocessor organization and establish performance metrics. We then calculate fixed area overheads for I/O logic/pads and the system bus. Next, we determine candidate system organizations and discuss our modeling approach. We then determine processing power, cycle time, and total system performance for each configuration. Lastly, we present our conclusions and discuss possible future extensions tc our work.

2 System Organization and Performance Metrics

The overall architecture that we consider is a shared memory multiprocessor system containing n processors, each of which has a private cache. A system bus is used as the interconnect structure between the caches and the main memory, which may consist of several interleaved modules. With the technology predicted to be available at the end of this decade, designers will be able to place this entire structure (for a limited number of processors and with the exception of the main memory DRAMs), onto a single chip[1].

The execution environment that we assume is a coarse-grain, *throughput-oriented*, parallel environment. The system might be for example a server running Unix with many X terminals connected to it. Each of these X sessions runs separate user tasks and applications, and shares primarily operating system code and data structures and common application code. Thus the amount of shared data is minimal, and the vast majority of the time the processors are accessing private data.

The performance metric that we wish to maximize is *total system performance* which can be expressed as *processing power/cycle time* where *processing power* is defined by $n/(CPI \cdot instr)$. Here n is the number of processors in the system, *CPI* is the cycles per instruction rating for each processor, and *instr* is the number of instructions executed in the running of the program. This last parameter is a function of the instruction set architecture. Since this paper does not focus on instruction set architecture design issues, we eliminate this term and thus obtain *n/CPI* for processing power.

3 Tradeoffs Between Cache Size and Number of Processors and the Effect of TLB Size

Having established the general system organization, execution model, and performance evaluation criteria, we now examine some of the tradeoffs involved in designing single chip multiprocessors. We consider the problem of whether to use the available chip area for enlarging the caches or for adding additional processors. We focus on these two architectural parameters (the size of the caches and the number of processors) since they have such a great impact on performance. Once we have made area tradeoffs for these parameters and have a region of the design space narrowed down, we can then address parameters which have less impact on performance. In this paper, we address the size of the TLB.

The physical aspects of our study are based on the dimensions of the Alpha 21064 microprocessor[5, 13]. We scale the processor core and cache dimensions of the 21064 to 0.25 micron technology, and assume the use of a one inch square die. We then use this data to obtain candidate multiprocessor configurations which vary in cache size and number of processors.

[1] An experimental version of such a chip has already been designed and fabricated in 0.30 micron technology[10].

3.1 Bus and I/O Overhead

We use an aggressive bus design, consisting of separate 64-bit wide address and 128-bit wide data buses distributed to all the caches as well as the main memory controller on the chip. Based on previous implementations of bus-based multiprocessors, an aggressive bus design is necessary for a moderately large (8-16) number of processors. We allocate roughly 20% of the chip area for the bus, external interface, and I/O pads. Based on empirical measurements, approximately 17% of the area of the 21064 is allocated to external control and I/O pads. Due to the large number of I/O and power signals expected on our chip, and because I/O pad size may not scale as well as transistor sizes, we assume the same overhead on our chip. To determine the area consumed by the bus, we scale the dimensions of the second layer of metal (Metal 2) on the Alpha chip, since this is the wider of two metal layers used for general signal distribution. (The third layer is primarily used for power and clock distribution.) Metal 2 has a width of $0.75\mu m$ and a pitch of $2.625\mu m$. We consider two means of scaling for comparison purposes: ideal scaling and $\sqrt{S_c}$ scaling where S_c is the scaling factor. The latter has been suggested in [1] in order to reduce propagation delays as technology is scaled. In our case, since we are scaling a 0.75 micron technology to 0.25 microns, S_c has a value of 3. We assume an additional 15 signals beyond those for address and data for arbitration, signaling of operations on the bus, and acknowledgements. Thus our total bus signal count is 207. We obtain for the width of the bus using ideal scaling $(0.75 \cdot 10^{-3} + 2.625 \cdot 10^{-3}) \cdot 207/3 = 0.23\text{mm}$ and using $\sqrt{S_c}$ scaling $(0.75 \cdot 10^{-3} + 2.625 \cdot 10^{-3}) \cdot 207/\sqrt{3} = 0.40\text{mm}$.

Assuming the bus runs almost the entire length (20mm) of the chip, it consumes 0.7% and 1.2% of the chip area with ideal and $\sqrt{S_c}$ scaling, respectively. Scaling the metal dimensions for the bus by $\sqrt{S_c}$ instead of ideally has a negligible impact on the overall chip area. Adding the area for I/O gives us 17.7% and 18.2% for ideal and $\sqrt{S_c}$ scaling, respectively. In order to account for additional area lost due to the routing problems inherent in such a wide bus, we boost our overall figure for the area consumed by the I/O interface and bus to 20%.

3.2 Candidate Configurations

To determine candidate configurations, we proceed as follows. First, we empirically determine the area of the caches (16kB total) and processor core (the chip minus the caches and I/O pads) of the 21064. We scale (using ideal scaling) these dimensions to 0.25 micron technology, and then determine the fraction of a 0.25 micron, one inch square die consumed by the scaled processor and caches. The area of 32kB, 64kB, and larger caches is determined by using multiples of the base 16kB cache area. This is to a first order, consistent with area models described in [14, 15], especially for large caches where the size of the data area dominates the overall size.

We combine the dimensions of these caches with the dimension of the processor core to construct candidate processor/cache organizations. We then determine how many of these can be placed on the chip. For smaller configurations, we determine this by dividing 80% of the total chip area by the processor/cache area. As the number of processors grows, we assume more area overhead (a few additional percent of the total chip area) is required for routing.

Based on this method, we obtain the candidate system organizations shown in Table 1. These provide a good range of design points for an initial analysis. Once we have analyzed these design points, we can make further refinements to arrive at alternative organizations.

Configuration	Number of Processors	Cache Size
1	7	256kB
2	11	128kB
3	16	64kB
4	20	32kB

Table 1: Candidate System Organizations

Cache Size	Miss Rate	Misses Causing Displacements	Displacement Buffer Hits
32kB	2.05%	23.44%	3.96%
64kB	1.33%	27.54%	3.92%
128kB	0.89%	30.85%	4.37%
256kB	0.55%	33.91%	3.74%

Table 2: Cache Simulation Results

We choose to bound the cache sizes at 256kB and 32kB. For our benchmarks, caches larger than 256kB produce diminishing returns in terms of miss rate; further increasing the size of the cache beyond 256kB at the expense of the number of processors clearly results in worse overall performance. Caches of size less than 32kB on the other hand, have high enough miss rates to saturate even a very wide data bus. Even the 256kB and 32kB caches are suspect in terms of these criteria; we include them in order to avoid inadvertently excluding the optimum configuration.

3.3 Modeling Approach

Due to our execution model assumptions, we use uniprocessor trace driven simulation to analyze the various cache options, and apply the results to our multiprocessor analytical models. The traces we use are of the SPEC KENBUS program running on an i486 processor under the MACH 3.0 operating system. These traces are from the BYU Address Collection Hardware (BACH) system[8] and include both user and operating system references. Our trace length is approximately 80 million references, and we ignore cold start effects. The results of our simulations are given in Table 2. Besides miss rate, we also gather information on the fraction of misses that cause a dirty block to be displaced from the cache, and the fraction of cache misses that hit in the four entry displacement buffer. This buffer operates as a FIFO and holds recently displaced blocks to reduce thrashing effects in our direct-mapped caches as described in [11]; our hit rate results are in agreement with this previous work.

The results from Table 2 are used as inputs to analytical models of our candidate organizations. We use Mean-Value-Analysis (MVA), an analytical modeling technique which has been used extensively to study shared memory multiprocessors[4, 18, 19]. Our model assumptions are given in Table 3. CPI_{proc} is the CPI rate of the processor with no cache misses. Our block size choice has been shown in [6] to be a reasonable choice for the KENBUS benchmark for caches in our range. The bus penalties assume that a "dead cycle" is needed to switch bus masters to avoid driver clashing.

Parameter	Value
CPI_{proc}	1.5
% loads	25%
% stores	10%
Block size	32 bytes
Disp buffer hit penalty	2 cycles
Bus read penalty	2 cycles
Bus writeback penalty	3 cycles
Bus data return penalty	3 cycles
Bus arbitration time	2 cycles
Memory access time	15 cycles

Table 3: MVA Model Parameters

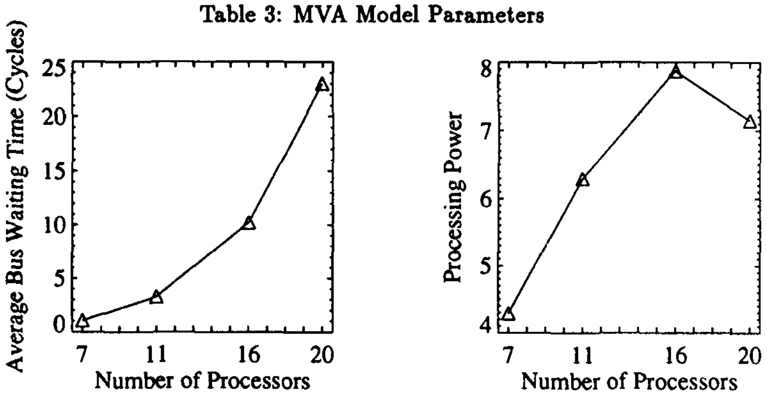


Figure 1: Average Bus Waiting Time and Processing Power for Candidate Configurations

3.4 Processing Power Results

Figure 1 shows the average bus waiting time and processing power for each of the candidate organizations. The larger cache (fewer processor) configurations as expected display very low waiting times, while the waiting time increases quickly for the smaller cache configurations. Thus we conclude that up to a point, adding processors at the expense of cache size is a good tradeoff. We see however that the 20 processor, 32kB cache configuration is clearly not a good design point to consider. The drop in processing power from the 16 processor, 64kB configuration is due to system bus saturation, and the resulting increase in waiting time cancels out the benefits received from adding processors.

The 16 processor, 64kB design point is suspect as well, although it provides the highest processing power. The increase in bus waiting time in this configuration as compared to the 11 processor, 128kB configuration suggests that this design point may not be optimal. It may be beneficial to reduce the number of processors in order to alter the processor organization. One option would be to increase the size of each processor's TLB while reducing the number of processors by an amount commensurate with the resulting area increase. This would result in fewer cache misses and a subsequent reduction in bus waiting time. In order to assess this impact, we use the results of area models developed

Cache Size	Miss Rate	Misses Causing Displacements	Displacement Buffer Hits
32kB	1.72%	24.24%	4.00%
64kB	1.12%	29.50%	4.34%
128kB	0.77%	33.27%	4.68%
256kB	0.49%	35.90%	3.94%

Table 4: Results of Cache Simulations with 512 Entry TLB

for TLBs and caches[14, 15] to assess the relative area of a 32 entry TLB (used in the i486 from which the traces were gathered) and a 64kB cache. From [15], these area estimates are roughly 2500 and 375000 rbes (register bit equivalents), respectively. A 512 entry TLB costs roughly 24000 rbes. Thus, the cost incurred in increasing the TLB from 32 entries to 512 entries is roughly 21500 rbes per processor. For a 15 processor configuration, this amounts to 322500 rbe or less than the cost of one 64kB cache. Thus, we see that by removing one processor and its cache, we can increase the TLB in each of the remaining 15 processors to 512 entries. This should reduce the amount of TLB misses by over 50% from that of a 32 entry TLB[3].

In order to get a lower bound on the impact of this architectural change on performance, we use a technique called *trace modification* to conservatively modify the trace to emulate a trace taken from a processor with a 512 entry TLB. To accomplish this, we use a program that marks the TLB misses in the trace[7]. We then make a conservative estimate as to the number of memory references in the TLB miss code. Our estimate is 35 references, a low estimate based on results of Mach TLB miss behavior[16]. We then insert a filtering program into our simulator that causes the simulator to ignore every other TLB miss in the trace. If a second TLB miss is encountered within 35 references of a filtered TLB miss, we filter it out as well since it results from the first TLB miss. Table 4 shows the results with the larger TLB. Comparing this table with Table 2, we see that for each cache configuration, the miss rate is lower with the larger TLB as expected.

We now use this data to evaluate the impact the larger TLB has on the bus waiting time. We look at various design points for the 64kB cache, with the number of processors ranging from 12 to 16. The larger TLB has a significant effect on the average bus waiting time, reducing it by 17-19%. When viewed from the perspective of equivalent multiprocessor configurations, that is, the number of processors with the larger TLB is one less than that with the 32 entry TLB, the impact is even greater, around a 28% reduction. Looking at these equivalent configurations in terms of processing power (Figure 2), we see that for the smaller configurations, those with the smaller TLB may still provide the best design points. However, we see that the 14 processor, 512 entry TLB and the 15 processor, 32 entry TLB configurations have roughly equivalent performance, and the 15 processor, 512 entry TLB configuration outperforms the 16 processor, 32 entry TLB configuration. Because we have obtained a lower bound on the increase in performance due to enlarging the TLB, the difference in performance may actually be larger than that shown.

3.5 Cycle Time Results

To examine cache access times, we make use of the model developed in [20] which is based on a 0.8 micron, 5V process. We use a voltage of 3.3V for our analysis, and scale the capacitance, resistance, and current parameters of this model to produce a 0.25

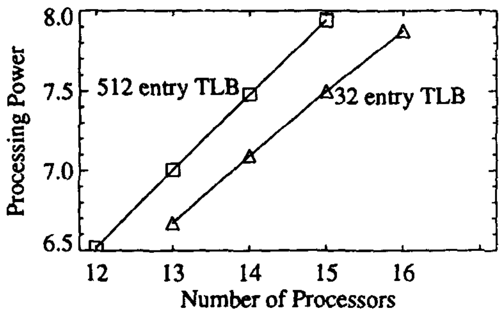


Figure 2: Processing Power for Equivalent 64kB Configurations

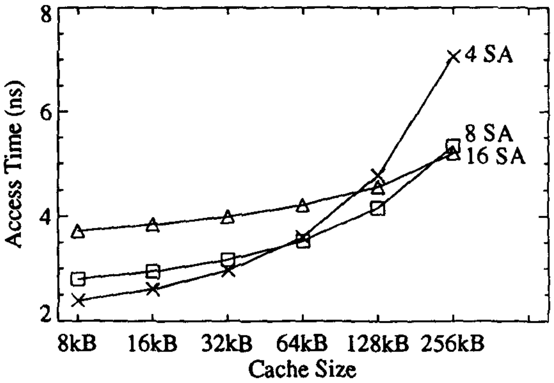


Figure 3: Cache Access Times for Various Sizes and Geometries

micron technology model[2]. We examine 4 (the same as the 21064), 8, and 16 subarray (SA) caches. Our results (Figure 3) indicate that caches larger than 64kB require more aggressive geometric design to achieve reasonable cycle times. This may result in more area overhead and less layout flexibility.

We now examine the effects of the number of processor nodes on bus performance. We assume equal space loading and since the total propagation delay is longer than the driver rise time, we use transmission line analysis. The loaded propagation delay of a transmission line is[2] $T_0 \cdot \sqrt{1 + C_D/C_0}$ where T_0 and C_0 are the unloaded transmission line propagation delay and capacitance, respectively, and C_D is the distributed capacitance due to each processor node. C_D can be expressed as $C_N \cdot s/l$ where C_N is the node capacitance, s is the rumber of line segments (one less than the number of nodes), and l is the length of the bus (20mm in our example). Following [2], we use C_N = 5fF, C_0 = 2pF/cm, and T_0 = 0.067ns/cm, and obtain the results shown in Figure 4. We conclude that shared buses can still achieve good cycle time performance for moderate numbers of processors. However, to achieve aggressive clock rates with large numbers of processors,

[2] We note that these models are highly dependent on technology parameters, and thus our method should only be used to study general trends and not exact cycle time analysis.

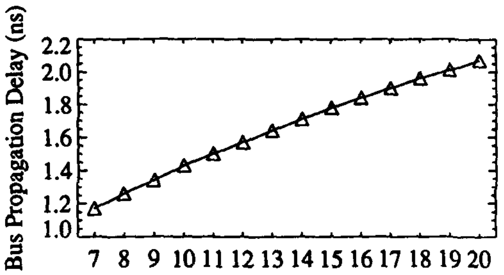


Figure 4: Bus Propagation Delay Variation with Number of Processors

Configuration	PP	Cycle Time		Total System Performance	
(Proc, Cache, TLB)		Cache Constrained	Bus Constrained	Cache Constrained	Bus Constrained
7, 256kB, 32 entry	1.00	1.00	1.00	1.00	1.00
11, 128kB, 32 entry	1.47	0.80	1.28	1.84	1.15
15, 64kB, 512 entry	1.85	0.68	1.52	2.72	1.22
16, 64kB, 32 entry	1.84	0.68	1.57	2.71	1.17

Table 5: Performance Data for Candidate Configurations

interconnect alternatives such as ring-based schemes[9] need to be considered.

3.6 Overall Performance

We now bring together the processing power and cycle time results and compare the overall performance of our candidate configurations. We use the 7 processor, 256kB configuration as our base system (and thus assign it unity performance values), and calculate processing power, cycle time, and total system performance for the other configurations relative to this organization. We look at two different cycle time scenarios: a *cache constrained* cycle time chip, and a *bus constrained* cycle time chip. In the former, the cache access time determines the cycle time of the chip; in the latter, the bus propagation delay determines this.

Table 5 shows the relative processing power (PP), cycle time, and total system performance for each of the candidate configurations. Due to their superior processing power performance, the 15 and 16 processor configurations achieve the best overall performance in both the cache and bus constrained cases. The 15 processor configuration is overall the best choice, as it provides the highest processing power, and an equal or faster cycle time than the 16 processor configuration. We note, however, that for a bus constrained cycle time chip, the performance of the 11 processor configuration closely matches that of the 15 and 16 processor configurations. For applications that consume more of the cache, this configuration may be the best choice. If an even number of processors (or a power of two) is more beneficial for the application software, then the number of processors can be reduced accordingly. This may require the designer to allocate more area to TLBs, interconnect, or other resources.

4 Conclusions and Future Work

The complexity of microprocessor design is growing rapidly as VLSI integration levels continue to increase. With the technology expected to be available at the end of this decade, microprocessor architects will be faced with a wide range of design decisions. An analysis of tradeoffs between the size of the caches and the number of processors in the system was presented. Our results show that trading off cache size for the number of processors improves performance up to the point of bus saturation. At this point, alternatives such as reducing the number of processors in order to increase TLB size, need to be considered in order to arrive at the optimal design point.

This work can be extended in several different ways. First of all, we assumed an independent execution model, whereby processors are accessing private code and data and negligible sharing takes place. We also looked at a single benchmark program. Examining the effect of a wider range of execution models (e.g., fine-grain parallel processing) and application programs on the design choices made would be insightful. Secondly, an examination of the sensitivity of our results to our model parameters (such as memory access time) should be made. Lastly, we only considered a single level of cache hierarchy. Multilevel cache hierarchies can be examined as well. The incorporation of multiple processors on a chip allows the designer to consider other cache options, such as private first level caches and multiported second level caches shared by two or more processors. Such an arrangement may make more efficient use of the available chip area than a conventional organization, and thus should be evaluated as well.

Acknowledgements

The authors wish to thank the members of the BACH project at Brigham Young University, especially Kelly Flanagan, Knut Grimsrud, and Brent Nelson, for providing the traces and the TLB miss marking tool used in this project. We also thank the anonymous referees for their helpful comments and suggestions.

References

[1] H.B. Bakoglu and J.D. Meindl, "Optimal Interconnection Circuits for VLSI," *IEEE Transactions on Electron Devices*, pp. 903-909, May 1985.

[2] H.B. Bakoglu, *Circuits, Interconnections, and Packaging for VLSI*, Addison-Wesley, Reading, MA, 1990.

[3] J.B Chen, A. Borg, and N.P. Jouppi, "A Simulation Based Study of TLB Performance," *19th International Symposium on Computer Architecture*, pp. 114-123, May 1992.

[4] M. Chiang and G.S. Sohi, "Evaluating Design Choices for Shared Bus Multiprocessors in a Throughput-Oriented Environment," *IEEE Transactions on Computers*, pp. 297-317, March 1992.

[5] D.P. Dobberpuhl, et al, "A 200MHz, 64-Bit, Dual-Issue CMOS Microprocessor," *Digital Technical Journal*, Vol. 4, No. 4, pp. 35-50, 1992.

[6] J.K. Flanagan, "A New Methodology for Accurate Trace Collection and Its Application to Memory Hierarchy Performance Modeling," PhD Dissertation, Brigham Young University, December 1993.

[7] K. Grimsrud, "Address Translation of BACH i486 Traces," Technical Memorandum, Brigham Young University, June 1993.

[8] K. Grimsrud, J. Archibald, M. Ripley, K. Flanagan, and B. Nelson, "BACH: A Hardware Monitor for Tracing Microprocessor-Based Systems," Technical Report, Brigham Young University, February 1993.

[9] D. Gustavson, "The Scalable Coherent Interface and Related Standards Projects," *IEEE Micro*, pp. 10-22, February 1992.

[10] M. Hanawa, et al, "On-Chip Multiple Superscalar Processors with Secondary Cache Memories," *International Conference on Computer Design*, pp. 128-131, October 1991.

[11] N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *17th International Symposium on Computer Architecture*, pp. 364-373, May 1990.

[12] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik, *Quantative System Performance, Computer Analysis Using Queuing Network Models*, Prentice Hall, Englewood Cliffs, N.J., 1991.

[13] E. McLellan, "The Alpha AXP Architecture and 21064 Processor," *IEEE Micro*, pp. 36-47, June 1993.

[14] J.M. Mulder, N.T. Quach, M.J. Flynn, "An Area Model for On-Chip Memories and Its Application," *IEEE Journal of Solid-State Circuits*, pp. 98-106, February 1991.

[15] D. Nagle, R. Uhlig, T. Mudge, and S. Sechrest, "Optimal Allocation of On-chip Memory for Multiple-API Operating Systems," *21st International Symposium on Computer Architecture*, April 1994.

[16] D. Nagle, R. Uhlig, T. Stanley, S. Sechrest, T. Mudge, and Richard Brown, "Design Tradeoffs for Software-Managed TLBs," *20th International Symposium on Computer Architecture*, pp. 27-38, May 1993.

[17] J. Rattner, "MICRO 2000: Microprocessors in the Year 2000," Keynote Address, Second Annual VLSI Educator's Conference, July 1989.

[18] M.K. Vernon, R. Jog, and G.S. Sohi, "Performance Analysis of Hierarchical Cache-Consistent Multiprocessors," *Performance Evaluation*, Vol. 9, pp, 287-302, 1989.

[19] M.K. Vernon, E.D. Lazowska, and J. Zahorjan, "An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache Consistency Protocols," *15th International Symposium on Computer Architecture*, pp. 308-315, June 1988.

[20] T. Wada, S. Rajan, S.A. Przybylski, "An Analytical Access Time Model for On-Chip Cache Memories," *IEEE Journal of Solid-State Circuits*, pp. 1147-1156, August 1992.

PART II

CODE GENERATION FOR MULTITHREADED AND DATAFLOW ARCHITECTURES

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

An Evaluation of Optimized Threaded Code Generation

Lucas Roh, Walid A. Najjar, Bhanu Shankar and A.P. Wim Böhm

Dept. of Computer Science, Colorado State University, Fort Collins, CO 80523, USA

Abstract: Multithreaded architectures hold many promises: the exploitation of intra-thread locality and the latency tolerance of multithreaded synchronization can result in a more efficient processor utilization and higher scalability. The challenge for a code generation scheme is to make effective use of the underlying hardware by generating large threads with a large degree of internal locality without limiting the program level parallelism. Top-down code generation, where threads are created directly from the compiler's intermediate form, is effective at creating a relatively large thread. However, having only a limited view of the code at any one time limits the thread size. These top-down generated threads can therefore be optimized by global, bottom-up optimization techniques. In this paper, we present such bottom-up optimizations and evaluate their effectiveness in terms of overall performance and specific thread characteristics such as size, length, instruction level parallelism, number of inputs and synchronization costs.

Keyword Codes: C.1.2; D.1.1; D.1.3
Keywords: multithreaded code generation; optimization; architectures; evaluation;

1 Introduction

Multithreading has been proposed as an execution model for large scale parallel machines. The multithreaded architectures are based on the execution of threads of sequential code which are asynchronously scheduled based on the availability of data. This model relies on each processor being endowed, at any given time, with a number of ready-to-execute threads to switch amongst them at relatively cheap cost. The objective is to mask the latencies of remote references and processor communication with useful computation and thus yielding high processor utilization. In many respects, the multithreaded model can be seen as combining the advantages of both the von Neumann and dataflow models: efficient exploitation of instruction level locality of the former and the latency tolerance and efficient synchronization of the later.

Thread behavior is determined by the firing rule. In a *strict* firing rule, all the inputs necessary to execute a thread to completion are required before the execution begins. On the other hand, a *non-strict* firing rule allows a thread to start executing when some of its inputs are available. In the latter case, threads can become larger, but the architecture must handle threads that block, with greater hardware complexity using pools of "waiting" and "ready" threads; examples include the HEP [1] and Tera machines [2]. The strict

This work is supported by NSF Grant MIP-9113268.

firing can be efficiently implemented by a matching mechanism using explicit token storage and presence bits; examples include MIT/Motorola Monsoon [3] and *T [4], and the ETL EM-4 and EM-5 [5].

A challenge lies in generating code that can effectively utilize the resources of multithreaded machines. There is a strong relationship between the design of a multithreaded processor and the code generation strategies, especially since the multithreaded processor affords a wide array of design parameters such as hardware support for synchronization and matching, register files, code and data caches, multiple functional units, direct feedback loops within a processing element and vector support. Machine design can benefit from careful quantitative analysis of different code generation schemes. The goal for most code generation schemes for nonblocking threads is to generate as large a thread as possible [6], on the premise that the thread is not going to be too large, due to several constraints imposed by the execution model. Two approaches to thread generation have been proposed: the *bottom up method* [7, 8, 9, 10] starts with a fine-grain dataflow graph and then coalesce instructions into clusters (threads), the *top down method* [11, 12, 13] generates threads directly from the compiler's intermediate data dependence graph form. In this study, we have combined the two approaches in which we initially generate threads top-down and then optimize these threads via bottom-up method. The top down design which suffers from working on one section of code at a time limits the thread size; on the other hand, the bottom up approach, with its need to be conservative, suffers from lack of knowledge of program structures thereby limiting the thread size.

In this paper, we compare the performance of our hybrid scheme with a top-down only scheme and measure the code characteristics and run-time performance. The remainder of this paper details the hybrid code generation scheme and the measurements and analysis of their execution. Due to space limitations, related work is not discussed in this paper. The reader is referred to [14, 15] for some background. In Section 2, we describe the code generation scheme. The performance evaluation is described in Section 3. Results are discussed in Section 4.

2 Thread Generation and Optimizations

In this section we describe the computation model underlying thread execution (section 2.1) and code generation (section 2.2). In Section 2.3 we describe the various optimizations techniques.

2.1 The Model of Computation

In our model, threads execute *strictly*, that is, a thread is enabled only when *all* the input tokens to the thread are available. Once it starts executing, it runs to completion without blocking and with a bounded execution time. This implies that an instruction that issues a split-phase memory request cannot be in the same thread as the instructions that use the value returned by the split-phase read. However, conditionals and simple loops can reside within a thread. The instructions within a thread are RISC-style instructions operating on registers. Input data to a thread is available in input registers which are read-only. The output data from a thread is written to output registers which are write-only. The order of instruction execution *within* a thread is constrained only by *true data dependencies* (dataflow dependencies) and therefore instruction level parallelism can be easily exploited by superscalar or VLIW processor. The construction of threads is guided by the following objectives: (1) Minimize synchronization overhead; (2) Maximize intra-thread locality; (3)

Threads are nonblocking; and (4) Preserve functional and loop parallelism in programs. The first two objectives seem to call for very large threads that would maximize the locality within a thread and decrease the relative effect of synchronization overhead. The thread size, however, is limited by the last two objectives. In fact, it was reported in [10] that blind efforts to increase the thread size, even when they satisfy the nonblocking and parallelism objectives, can result in a decrease in overall performance. Larger threads tend to have larger number of inputs which can result in a larger input latency (input latency, in this paper, refers to the time delay between the arrival of the first input to a thread instance and that of the last input, at which time the thread can start executing [16]).

2.2 Top Down Code Generation

Sisal is a pure, first order, functional programming language with loops and arrays. We have designed a Sisal compiler for multithreaded/medium-grain dataflow machines by targeting to machine independent dataflow clusters (MIDC). Sisal programs are initially compiled into a functional, block-structured, acyclic, data dependence graph form, called IF1, which closely follows the source code. The functional semantics of IF1 prohibits the expression of copy-avoiding optimizations.

IF2, an extension of IF1, allows operations that explicitly allocate and manipulate memory in a machine independent way through the use of buffers. A buffer comprises of a buffer pointer into a contiguous block of memory and an element descriptor that defines the constituent type. All scalar values are operated by value and therefore copied to wherever they are needed. On the other hand, all of the fanout edges of a structured type are assumed to reference the same buffer; that is, each edge is not assumed to represent a distinct copy of the data. IF2 edges are augmented with pragmas to indicate when an operation such as "update-in-place" can be done safely which dramatically improve the run time performance of the system.

The top down cluster generation process transforms IF2 into a flat MIDC where the nodes are clusters of straight line von Neumann code, and the edges represent data paths. Data travelling through the edges are tagged with an *activation name*, which can be interpreted as a stack frame pointer as in [17] or as a color in a classical dataflow sense.

The first step in the IF2 to MIDC translation process is the graph analysis and splitting phase. This phase breaks up the hierarchical, complex IF2 graphs so that threads can be generated. Threads terminate at control graph interfaces for loops and conditionals, and at nodes for which the execution time is not statically determinable. For instance, a function call does not complete in fixed time, neither does a memory access. Terminal nodes are identified and the IF2 graphs are split along this seam.

A function call is translated in code that connects the call site to the function interface, where the input values and return contexts are given a new activation name. The return contexts combine the caller's activation name and the return destination, and are tagged (as are the input values) with the new activation name. For an *if-then-else graph*, code for the *then* and *else* graphs is generated, and their inputs are wired up using conditional outputs. *Iterative loops* with loop carried dependences and termination tests, contain four IF2 subgraphs: *initialize, test, body* and *returns*. The returns graph produces the results of the loop. They are generated and connected sequentially, such that activation names and other resources for these loops can be kept to a minimum. *Forall loops* with data independent body graphs consist of a *generator*, a *body*, and a *returns* IF2 graphs. They are generated so that all parallelism in these loops can be exploited including out-of-order reduction. This is valid as Sisal reduction operators are commutative and associative.

```
define main
type OneDim = array[integer]
function main (list:OneDim returns double_real)
   for I in list
      returns value of sum 1.0d0/double_real(I) when I ~= 0
   end for
end function
```

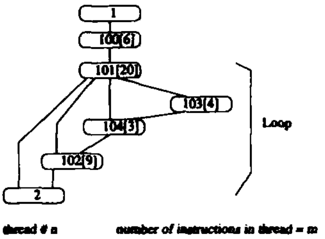


Figure 1: Purdue benchmark 4: Sisal code and MIDC graph

2.3 Bottom-up Optimizations

Even though the IF2 optimizer performs an extensive set of optimizations at the IF2 level including many traditional optimizations techniques such as function in-lining, the thread generation creates opportunities for more optimizations. We have applied optimizations at both the intra-thread and inter-thread levels. These optimizations are by no means exhaustive, with many more techniques that can be applied further. However, they do demonstrate its effectiveness.

Local Optimizations. These are traditional compiler optimizations whose main purpose is to reduce the number of instructions in a thread. They include: *Local dead code elimination*, *Constant folding/copy propagation*, *Redundant instruction eliminations* and *Instruction scheduling to exploit the instruction level parallelism* which is accomplished by ranking instructions according to dependencies such that dependent instructions are as far from each other as possible. The data and control flow representation allows this to be accomplished relatively easily.

Global Optimizations. The objectives of these optimizations are threefold: (1) Reduce the amount of data communication among threads; (2) Increase the thread size without limiting inter-thread parallelism; (3) Reduce the total number of threads. These optimizations consist of:

- *Global dead code and dead edge elimination:* typically applied after other global optimizations which could reduce the arcs or even the entire thread.

- *Global copy propagation/constant folding:* works to reduce unnecessary token traffic by bypassing intermediate threads.

- *Merging:* operations attempt to form larger threads by combining two neighboring ones while preserving the semantics of strict execution. This results in reduced synchronization cost. The merge up/merge down phases have been described in [7] and in [10]. In order to ensure that functional parallelism is preserved and bounded thread execution time is retained, merging is not performed across: remote memory access operations, function call interface and the generation of parallel loop iterations.

- *Redundant arc elimination:* opportunities typically arise after merging operation when arcs carrying the same data could be eliminated.

Even though the local optimizations could be performed during the thread generation, we have performed both local and global optimizations as a separate bottom-up pass which

```
; input function interface
F main 1 1 0 <(100 1)>

; output function interface
F main 2 4 0 < >
IN 1 3

; initialization. fetch
; array descriptors
N 100 2 1 < >
R1 = EIX
R2 = EAN
RSS I1 "1" "101D2" R1 R2
RSS I1 "2" "101D1" R1 R2
RSS I1 RO "101D3" R1 R2
RSS I1 "1" "101D4" R1 R2

; loop generator that spawns
; task
N 101 10 4 <(2 1 0)(102 2 0)
   (102 1 0)(103 3 0)(103 2 0)
   (103 1 0)(104 3 0)(104 2 0)>
R1 = EIX
R2 = EAN
R3 = ADI I1 I2
R4 = SUI R3 "1"
R5 = LEI I2 R4
```

```
      (cont'd)
IF R5
02 = OUT "0.0D0" R1 R2
R6 = SUI R4 I2
R7 = ABI R6
R8 = ADI R7 "1"
03 = OUT R8 R1 R2
R9 = DO I2 R4 "1"
R10 = GAN
04 = OUT I3 R9 R10
05 = OUT I4 R9 R10
06 = OUT R9 R9 R10
07 = OUT R2 R9 R10
08 = OUT R1 R9 R10
ENDDO
ELSE
01 = OUT "0.0D0" R1 R2
ENDIF
```

```
; handles the summation
; reduction
N 102 5 3 <(2 1 0)(102 2 0)
     (102 1 0)>
R1 = EIX
R2 = EAN
R3 = SUI I1 "1"
R5 = ADD I2 I3
R4 = EQI R3 RO
IF R4
01 = OUT R5 R1 R2
ELSE
02 = OUT R6 R1 R2
03 = OUT R3 R1 R2
ENDIF

; fetch an element
N 103 3 3 < >
R1 = EIX
R2 = EAN
R3 = SUI I1 I2
RSS I3 R3 "104D1" R1 R2

; invert the element
; value
N 104 2 3 <(102 3 3)>
R1 = ITD I1
R2 = DVD "1.0d0" R1
01 = OUT R2 I2 I3
```

Figure 2: MIDC thread code for Purdue benchmark 4

does extensive graph traversal and analysis. To support these operations, the optimizer pass reads MIDC and internally builds a mini-dataflow graph for each thread along with the global inter-thread interconnections. At this point, we have the complete, equivalent dataflow graph that a pure bottom-up compiler would generate but retains the additional *top-down structural informations*. When the optimized code is generated, each thread is generated from these mini-dataflow graphs. The global view enables various optimization steps to take place and enable redrawing of "boundaries" to easily recast threads.

A simple Sisal source and its compiled, optimized MIDC code is shown in Figures 1 and 2. Each node in MIDC has a header followed by the thread of instructions. The header has the form "N n r i < destination list >" where n specifies the unique thread number, r the number of registers the thread uses, and i specifies the number of inputs[1]. Each destination in the destination list corresponds to an output port to which the result is sent. The example code is taken from Purdue benchmark 4 which inverts each nonzero element of an array and and sums them. Figure 1 shows the thread level graph descriptions. Nodes 1 and 2 are the main function input and output interfaces, respectively. Node 100 reads the structure pointer and size information. Node 101 is the loop initializer/generator. Node 102 and 103 comprises the loop body and Node 104 handles the array reduction.

3 Evaluation

In this section we evaluate the dynamic properties of our top-down code after applying various bottom-up optimizations and using a multi-threaded machine simulator. The following set of *dynamic intra-thread statistics* have been collected: (1) S_1: average thread

[1] Complete details of the MIDC syntax are described in [15].

	Linpack			Purdue			Loops			Average		
	no	local	full	no	local	full	no	local	full	no	local	full
S_1	9.50	8.73	7.83	12.52	11.85	11.08	23.09	21.01	19.71	14.73	13.66	12.52
S_∞	2.47	2.47	2.48	2.99	2.98	3.09	3.69	3.63	3.92	3.07	3.05	3.17
Π	3.70	3.45	3.01	4.44	4.24	3.67	7.08	6.68	5.48	4.99	4.73	3.98
Input	4.88	4.88	3.73	5.78	5.78	4.45	12.16	12.16	9.63	7.29	7.29	5.59
MPI	0.44	0.47	0.40	0.46	0.49	0.40	0.51	0.56	0.47	0.48	0.52	0.43

Table 1: Intra-Thread Characteristics

size; (2) S_∞*:* average critical path length of threads (in instructions; (3) Π*:* average internal parallelism of threads ($\Pi = \frac{S_1}{S_\infty}$); (4) ***Inputs:*** average number of inputs of threads; (5) *MPI:* number of matches (synchronizations) per instruction (computed by dividing the total number of matches by the total number of instructions executed).

Also, the following set of inter-thread (program-wide) statistics have been collected: (1) ***Matches:*** total number of matches performed; (2) ***Instructions:*** total number of instructions executed; (3) ***Threads:*** total number of threads executed; and (4) *CPL:* critical path length of programs (in threads). Values of these inter-thread parameters are presented in normalized form with respect to the un-optimized code.

The set of benchmarks includes: Lawrence Livermore Loops (LL 1 to 24), Purdue benchmarks (PB 1 to 15), and several Linpack benchmarks (LB). We use three levels of optimization: (1) *No optimization:* code generated by the MIDC compiler; (2) *Local optimization:* only at the intra-thread level; and (3) *Full optimization:* both intra-thread and global optimizations.

3.1 Intra-thread Results

Table 1 shows the thread characteristics for the three classes of benchmarks and the cumulative of all the benchmarks. The values shown are weighted average values for a given class of benchmarks.

Thread Size. Results show a steady reduction of the average size of threads as we go from no optimization to local to full optimizations. The reduction occurs despite the merging operations. This is due to large reduction in the total number of instructions executed. The distribution of S_1 (Fig. 3) shows that 34% of all threads have six instructions. Threads with $S_1 \geq 10$ account for 23% of total.

Thread Critical Path Length. A slight reduction in S_∞ occurs in going from no optimization to local optimization. However, there is a nontrivial increase in in the path length from un-optimized code to fully optimized code. This is especially true for LL and PB. The distribution of S_∞ (Fig. 4) shows a strong dominance of thread lengths of two, three and four accounting for 85% of all threads. This indicates that when the instruction level parallelism within a thread is exploited, the thread execution time could be very short.

Thread Parallelism. The trend for Π is similar to S_1 in that parallelism decreases as more optimizations are applied in all benchmarks. Overall, due to the reduction of thread size, Π decreases even though S_∞ increases as more optimizations are applied. In fully optimized cases, average parallelism range from about 3.0 to 5.5 with the average of 4. There is a relatively wide variances in the parallelism. The intra-thread parallelism is large enough to justify a highly superscalar processor implementation. The distribution

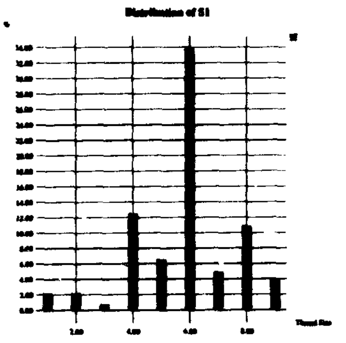


Figure 3: Distribution of S_1 ($S_1 \leq 9$)

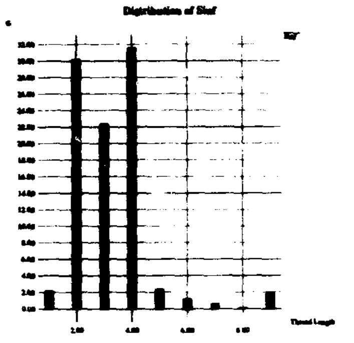


Figure 4: Distribution of S_∞ ($S_\infty \leq 9$)

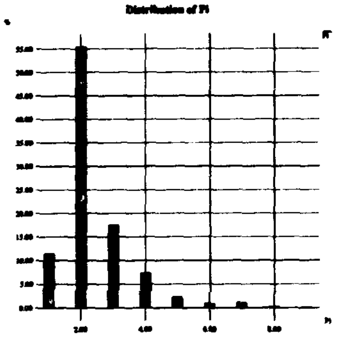


Figure 5: Distribution of Π ($\Pi \leq 9$)

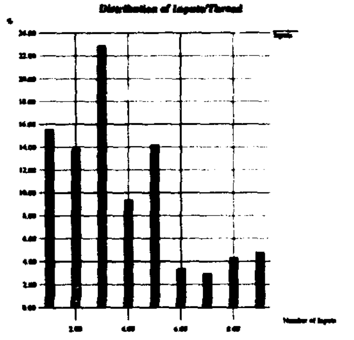


Figure 6: Distribution of inputs ($Inp \leq 9$)

of Π (Fig. 5) is dominated by $\Pi = 2$. An instruction level parallelism of four within a processor is sufficient for 92% of all threads executed.

Thread Inputs. Intra-thread optimizations do not affect the number of inputs per thread. However, there is a significant reduction of about 24% in the number of inputs after global optimizations. The average number of inputs varies widely between different benchmarks: LL has the largest number of inputs (9 to 12) and LB the smallest (4 to 5). With full optimizations, the cumulative average is about 5.6 inputs. In the distribution of the number of inputs per thread (Fig. 6), the dominant value is three inputs per thread. Threads with eight or less inputs account for 86.5% of all executed threads across all benchmarks.

Matches Per Instruction. For a given optimization level, the matches per instructions are confined within a narrow range for all classes of benchmarks. The MPI goes up when local optimizations are applied; MPI goes down by 8 to 13% when fully optimized. This implies that there are more significant reduction in the number of matches than the reduction in the number of instructions, resulting in a smaller MPI for the fully optimized code.

	Linpack		Purdue		Loops		Average	
	local	full	local	full	local	full	local	full
Match	1	0.76	1	0.68	1	0.67	1.00	0.69
Instruction	0.93	0.82	0.9	0.78	0.91	0.73	0.93	0.77
Thread	1	0.99	1	0.88	1	0.85	1.00	0.89
CPL	1	0.99	1	0.81	1	0.77	1.00	0.83

Table 2: Program Characteristics (Normalized to Un-optimized Code)

3.2 Program-wide Characteristics

In Table 2, the inter-thread characteristics of programs are shown. The values shown are normalized with respect to the un-optimized code.

Matches. As would be expected, the number of matches remain unchanged when only local optimizations are applied. When global optimizations are also applied, there is a significant reduction in the number of matches performed, typically 24% to 33%.

Instructions. There is on the average a 7% reduction in the total number of instructions executed when local optimizations are performed. After applying global optimizations, the average reduction is about 23%.

Threads. The number of threads executed remain unchanged for locally optimized code. When full optimizations are applied, there is a 10% reduction in the number of threads on the average. The reduction for the LL and PB are 12 to 15% but the reduction for the LB are very small, a little more than 1%.

Critical Path Length. The characteristics of critical path lengths of programs are very similar to the number of threads executed. The path lengths are reduced by about 20% for the LL and PB but for the LB, the reduction is very small. While the CPL is reduced by 17% the total number of threads is reduced by only 11%: this implies that, on average, more threads on the critical path length are eliminated than otherwise. This implies an increase in the inter-thread parallelism.

3.3 Run-time Performance

The results of simulated real time performance applied to different optimization levels are used to measure the effects of these optimizations on the actual overall performance.

The results are obtained with the following architectural settings: instruction level parallelism that a processor can utilize is 4, the output bandwidth, which is the number of tokens that a processor node can put on the network in a single cycle is 2, and a memory access latency of 5 cycles. We use the Motorola 88110 timing for instruction execution times. All the tokens go through the network at a cost of 50 cycles except in the uniprocessor setting. We obtained results for 1, 100, and infinite number of processors. These architectural features are not meant to model an existing system exactly, but rather approximate it to give us an empirical idea on what we can expect. In each case, the run time performances for both locally and fully optimized codes are compared with respect to the un-optimized code.

Table 3 shows the percentage improvement in speedup over the un-optimized code. The improvement in speed is significant with most improvement occurring when global optimizations are applied. With local optimization, only a single digit percentage speedup is achieved in most cases. Global optimizations can achieve up to 35% speedup in the case of LL. For LB and LL, the speedup remains relatively independent of the number of

	P=1		*P=100*		*P=∞*	
Benchmarks	local	global	local	global	local	global
Linpack	6.0	22.8	16.5	21.7	17.3	21.6
Purdue	2.0	19.6	4.8	26.3	5.5	27.6
Loops	3.4	35.1	8.2	33.7	8.7	34.8
Average	3.1	28.4	7.9	29.9	8.4	30.9

Table 3: Percent Speedup over Un-optimized Code

processors. However, for PB, the speedup rises as the number of processors increase, this is due to their larger average program parallelism.

4 Discussion and Conclusion

In this paper we have described and evaluated a top-down thread generation method and a set of local and global code optimizations. The effects of these on the intra-thread (local) and inter-thread (global) characteristics have been evaluated. The initial threaded code is generated from Sisal via its intermediate form IF2. While the statistics on intra-thread and program level parameters are important to understand the behavior of the thread generation and on the expected resource requirements, the bottom-line of any performance measure is the overall program execution time. In this respect, the local optimizations had a relatively small but meaningful effect while the global ones had close to a 30% reduction in execution time.

Overall, the average size of threads is relatively large compared to some of the values reported for bottom-up thread generation techniques [7, 14]. The threads in LL are in general much larger than the other classes of benchmarks. This is due to compact loop kernels whose inner loops consist of only a few threads. For the LB, the global optimization only minimally affects the number of threads executed or the critical path length of programs. This is due to the small size of these benchmarks which does not give much opportunities for the merge operations. Nevertheless, some speed-up is still observed.

The internal thread parallelism achieved is surprisingly large compared with our previous work on the evaluation of the bottom-up Manchester cluster generations which only achieved parallelism of about 1.15-1.20. We observe that the number of inputs required are relatively large, on the order of 4 to 10. This implies a need fcr handling these variable number of inputs efficiently.

In general, we observe that even though there is some improvement in various measures when local-only optimizations are applied, the biggest improvement comes from global optimizations. The results of the bottom-up optimizations indicate that they have achieved our objectives in code generation: (1) Synchronization overhead (e.g. number of matches) has been reduced significantly. The total number of matches has been reduced by 25-30% and *MPI* has been reduced by about 10%, thus reducing the required synchronization bandwidth. (2) Number of inputs to a thread has been reduced by about 24%, thus reducing the input latency. (3) Internal parallelism has been reduced by about 25%, meaning that processor requirements are not as great as before.

It should be noted there are many other techniques such as *loop un-rolling* that could be incorporated. For example, loop un-rolling can be applied either within a thread or across threads. The expected effects of loop un-rolling are: (1) an increase in the average thread size, (2) a decrease in the relative cost of synchronizations (matches per instruction), (3) a decrease in the total dynamic thread count. This would translate in an increased ability to mask latency and a reduced synchronization overhead albeit at the cost of increased demands on the instruction level parallelism within a processor.

References

[1] B. J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE (Real Time Signal Processing*, 298:241–248, 1981.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Portfield, and B. Smith. The Tera computer system. In *Proc. Int. Conf. on Supercomputing*, pages 1–6. ACM Press, June 1990.

[3] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *Int. Ann. Symp. on Computer Architecture*, June 1990.

[4] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Int. Ann. Symp. on Computer Architecture*, 1992.

[5] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a data-flow single chip processor. In *Int. Ann. Symp. on Computer Architecture*, pages 46–53, May 1989.

[6] R. A. Iannucci. Toward A Dataflow/Von Neumann Hybrid Architecture. In *Int. Ann. Symp. on Computer Architecture*, pages 131–140, 1988.

[7] K. E. Schauser, D. E. Culler, and T. von Eicken. Compiler-controlled multithreading for lenient parallel languages. In J. Hughes, editor, *Conf. on Functional Programming Languages and Computer Architecture*, 1991.

[8] J. E. Hoch, D. M. Davenport, V. G. Grafe, and K. M. Steele. Compile time partitioning of a non-strict language into sequential threads. Technical report, Sandia National Laboratory, 1992.

[9] K. R. Traub. Multithread code generation for dataflow architetures from non-strict programs. In J. Hughes, editor, *Conf. on Functional Programming Languages and Computer Architecture*, 1991.

[10] L. Roh, W. A. Najjar, and A.P.W. Böhm. Generation and quantitative evaluation of dataflow clusters. In *Conf. on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, June 1993.

[11] V. Sarkar. Partitioning and scheduling parallel programs for execution on multiprocessors. Technical Report CSL-TR-87-328, Stanford University, Computer Systems Laboratory, April 1987.

[12] L. Bic, M. Nagel, and J. Roy. Automatic data/program partitioning using the single assignment principle. In *Proc. Supercomputing '89*, Reno, Nevada, 1989.

[13] M. Girkar and C. D. Polychronopolous. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):166–177, March 1992.

[14] W.A. Najjar, L. Roh, and A.P.W. Böhm. The initial performance of a bottom-up clustering algorithm for dataflow graphs. In *Proc. of the IFIP WG 10.3 Conf. on Architecture and Compilation Techniques for Medium and Fine Grain Parallelism*, Orlando, FL, January 1993.

[15] A. P. W. Böhm, W. A. Najjar, B. Shankar, and L. Roh. An evaluation of bottom-up and top-down thread generation techniques. In 26th *ACM/IEEE International Symposium on Micrtoarchitecture (MICRO-26)*, 1993.

[16] W. A. Najjar, W. M. Miller, and A. P. W. Böhm. An analysis of loop latency in dataflow execution. In *Int. Ann. Symp. on Computer Architecture*, Gold Coast, Australia, May 1992.

[17] D. E. Culler and G. M. Papadopoulos. The explicit token store. *Journal of Parallel and Distributed Computing*, 10(4), 1990.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Functional, I-Structure, and M-Structure Implementations of NAS Benchmark FT

S. Sur and W. Böhm

Department of Computer Science
Colorado State University
Ft. Collins, CO 80523

Abstract:

We implement the NAS parallel benchmark FT, which numerically solves a three dimensional partial differential equation using forward and inverse FFTs, in the dataflow language Id and run it on a one node monsoon machine. Id is a layered language with a purely functional kernel, a deterministic layer with I-structures, and a non-deterministic layer with M-structures. We compare the performance of versions of our code written in these three layers of Id. We measure instruction counts and critical path length using the Monsoon Interpreter Mint. We measure the space requirements of our codes by determining the largest possible problem size fitting on a one node monsoon machine. The purely functional code provides the highest average parallelism, but this parallelism turns out to be superfluous. The I-structure code executes the minimal number of instructions and as it has a similar critical path length as the functional code, runs the fastest. The M-structure code allows the largest problem sizes to be run at the cost of about 20% increase in instruction count, and 75% to 100% increase in critical path length, compared to the I-structure code.

Keyword Codes: D.1.1.; D.3.3.; E.2
Keywords: Applicative Programming, Language Constructs and Features, Data Storage Representations

1 Introduction

In this paper we study the design of efficient declarative programs by implementing the NAS three dimensional FFT PDE benchmark FT [2] in Id [8]. This study is part of a larger project where we try to assess which declarative language features are of importance to write efficient scientific codes. A declarative programming language allows expressing *what is to be done*, without specifying too much of *how it is to be done*. As an example, declarative programming languages are implicitly parallel, i.e., allocation of tasks and data on processors are not expressed in the program. This frees the programmer from this level of complexity in the design of parallel programs.

The declarative programming language Id has a functional kernel. Arrays in this functional kernel are created using monolithic array constructors called *array comprehensions.* In [1], Arvind and others argue that these array comprehensions lack expressiveness, and that for certain problems a lower level of constructs, manipulating the elements of I-structures, is necessary. An I-structure is a single assignment array with element-level synchronization for reads and writes. Because of their single assignment nature, Id programs with I-structures are deterministic, even though pure functional referential transparency has been lost. In [3] it is shown that for certain problems I-structures are again not powerful enough and that the more expressive M-structures are needed. M-structures are also arrays with element-level synchronization, but do not have the single assignment property anymore. M-structures allow "put" operations to write in an empty array slot, and "get" operations read and empty an array slot. Therefore, with interleaved puts and gets, M-structures allow destructive updates to express potentially non-deterministic producer-consumer relationships.

The above papers argue convincingly that the Id language gets increasingly more expressive when I-structures and M-structures are added. In this paper, we are interested in the time and space efficiency of realistic programs, when written in these various layers of the language. We therefore analyse three Id implementations of the NAS FT benchmark: a functional version, a version using I-structures, and a version using M-structures. We measure the time complexity of our programs by running small problems on the Monsoon Interpreter MINT, which reports on the number of instructions executed and the critical path length and provides parallelism profiles. We measure the space complexity of our programs by determining the maximal problem size that fits in our one node Monsoon machine [7], which has a 4 Megaword data memory. In this case, the goal is to run a 64 × 64 × 64 problem, given in the NAS benchmark specification as the minimal sized problem, on a one node Monsoon machine. One 3-D 64^3 object contains half a Megaword of floating point numbers.

It turns out that the purely functional code provides the highest parallelism, but at the cost of high instruction counts and high space usage. The I-structure code executes the minimal number of instructions and runs the fastest on the one node Monsoon machine, which provides 8-fold parallelism. The M-structure code allows the largest problem sizes to be run and turns out to be the only code that allows us to run the $64 \times 64 \times 64$ problem.

The rest of this paper is organized as follows. Section 2 defines the FT NAS benchmark solver. Section 3 first discusses the representation of 3-D objects, and then highlights the differences in programming styles in the functional, I-structure, and M-structure codes. In section 4 we analyse the time and space performance of our three codes. Section 5 mentions related work and concludes.

2 Problem specification

In the NAS benchmark *FT*, the following three dimensional heat equation is solved numerically:

$$\frac{\delta u(x,t)}{\delta t} = \alpha \nabla^2 u(x,t)$$

where x is a position in three dimensional space and α a constant describing conductivity. When a Fourier transform is applied to each side, this equation becomes:

$$\frac{\delta v(z,t)}{\delta t} = -4\alpha\pi^2 |z|^2 v(z,t)$$

where $v(z,t)$ is the Fourier transform of $u(x,t)$. This equation has the solution:

$$v(z,t) = e^{-4\alpha\pi^2 |z|^2 t} v(z,0)$$

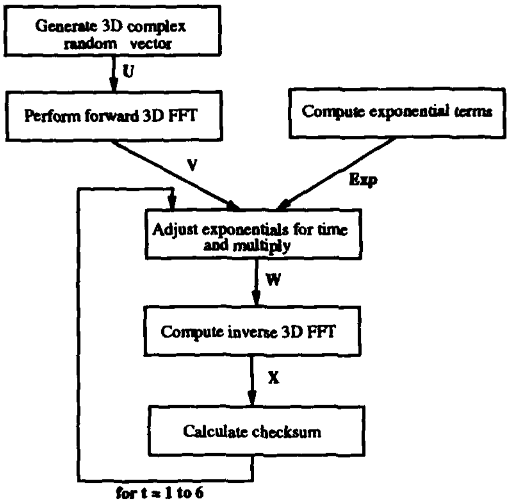

Generate 3D complex random vector
U
Perform forward 3D FFT
Compute exponential terms
V
Exp
Adjust exponentials for time and multiply
W
Compute inverse 3D FFT
X
Calculate checksum
for t = 1 to 6

Figure 1: Flow diagram of top level of NAS benchmark FT

The discrete version of the above problem can be solved using Discrete Fourier Transforms (DFT) instead of continuous ones. First, a 3-D DFT is performed on the original state array $u(x,0)$, then the results are multiplied by certain exponentials and lastly an inverse 3-D DFT is performed (see Figure 1.) The forward and inverse DFTs of the n_1 x n_2 x n_3 array u are defined respectively as:

$$F_{q,r,s}(u) = \sum_{l=0}^{n_3-1} \sum_{k=0}^{n_2-1} \sum_{j=0}^{n_1-1} u_{j,k,l} e^{-2\pi ijq/n_1} e^{-2\pi ikr/n_2} e^{-2\pi ils/n_3}$$

$$F^{-1}_{q,r,s}(u) = \frac{1}{n_1 n_2 n_3} \sum_{l=0}^{n_3-1} \sum_{k=0}^{n_2-1} \sum_{j=0}^{n_1-1} u_{j,k,l} e^{2\pi ijq/n_1} e^{2\pi ikr/n_2} e^{2\pi ils/n_3}$$

In the FT benchmark, the complex array U is initialized using a pseudo-random number generator. Setting V equal to the 3-D DFT of U, $\alpha = 10^{-6}$ and $t = 1$, the intermediate value W is computed:

$$W_{j,k,l} = e^{-4\alpha\pi^2(\bar{j}^2+\bar{k}^2+\bar{l}^2)t} V_{j,k,l}$$

where $\bar{j}$ is defined as j for $0 \leq j < n_1/2$ and $j - n_1$ for $n_1/2 \leq j < n_1$. The indices $\bar{k}$ and $\bar{l}$ are similarly defined with n_2 and n_3. X, the 3-D inverse DFT of W, is then computed. Finally, a checksum $\sum_{j=0}^{1023} X_{q,r,s}$ is computed where $q = j \ (mod\, n_1)$, $r = 3j \ (mod\, n_2)$ and $s = 5j \ (mod\, n_3)$. The computation of W, X and the checksum, is repeated for values t from 2 to 6. V needs only to be computed once. The array of exponential terms for $t > 1$ can be obtained as the t-th power of the array for $t = 1$.

The benchmark allows any algorithm be used for the computation of the 3-D FFTs. The algorithm we implement takes a complex array of size $n_1 \times n_2 \times n_3$ and performs $n_2 \times n_3$ n_1-point 1-D FFTs in the $n1$ direction, followed by $n_3 \times n_1$ n_2-point 1-D FFTs in the $n2$ direction, followed by $n_1 \times n_2$ n_3-point 1-D FFTs in the $n3$ direction yielding the final result. The benchmark also allows any algorithm to be used for the individual 1-D complex FFTs. We use a straight-forward iterative algorithm which reorders the array using bit-reversal of the index, and performs butterfly group recombinations, where the smallest group is 4, and the group size doubles in each iteration. Using a bottom case of 4 instead of 1 or 2 cuts out the bottom-most branches of the FFT tree making it much more space and time efficient. It also makes resource management simpler as pointers and objects are not interchanged [4]. We use iterative FFTs instead of recursive ones, because in the iterative codes all the intermediate arrays can be deallocated immediately after they are no longer required. Details about this can also be found in [4].

3 Implementation

3.1 Data Representation

The first choice for the data representation that comes to mind is a 3-D array of complex numbers represented by tuples of two real numbers. However, Id does not treat, for instance, a 1-D sub-array of such an array (e.g. A[i,j,*]) as an independent data structure. Ideally we would like a 1-D FFT function to be able to generate its result directly in a 1-D sub-array of our 3-D array in any of the 3 directions. As this is impossible, it is just as simple and more efficient to have a linear data structure representing the 3-D object. Selecting a vector in a certain direction now becomes stepping through the array with the appropriate stride. Having the complex numbers represented by tuples introduces considerable inefficiency because of the extra indirection introduced by the tuples. Moreover, deallocating an array of tuples can cause complications, if the array elements are sometimes copies (in which case only a new pointer is created) and sometimes new values (in which case a new tuple and a new pointer is created) [4]. We therefore opt for the simplest data representation possible: a linear array of $2n_1n_2n_3$ floating point numbers. To get the input array, $2n_1n_2n_3$ pseudo-random floating point values are generated as specified in the FT benchmark, and then used to fill the complex array $U_{j,k,l}$, $0 \leq j < n_1$, $0 \leq k < n_2$, $0 \leq l < n_3$, where the first dimension varies most rapidly as in the ordering of a 3-D Fortran array. A single complex entry of U consists of two consecutive pseudorandomly generated results and is stored such that the real and imaginary parts are a distance of $n_1n_2n_3$ apart.

3.2 Purely functional implementation

In the functional version of the program, arrays are created using *array comprehensions.* An array comprehension creating an n-dimensional array consisting of m regions is of the form:

$$\begin{array}{l} nD\_array((l_1,u_1),\ldots,(l_n,u_n))\text{ of} \\ \quad \mid [f_1^1(I_1),\ldots,f_1^n(I_1)] = expr_1 \;\; \| \; gen_1^1 \;;\ldots;\; gen_1^{d_1} \\ \quad \vdots \\ \quad \mid [f_m^1(I_m),\ldots,f_m^n(I_m)] = expr_m \;\; \| \; gen_m^1 \;;\ldots;\; gen_m^{d_m} \end{array}$$

where each generator gen_j^k , is of the form: $i_j^k \leftarrow l_j^k$ *to* u_j^k. Zero or more generator expressions define a region using a cross product of nested loops. I_j is the vector of loop variables for region j. Array comprehensions allow for elegant concise definitions of arrays. There are, however, a number of problems:

- **No Sharing.** When the computation of a number of array elements can share subcomputations, this cannot be expressed in an array comprehension, as each array element is defined independently.
- **No sub-array target.** We cannot create a substructure and scatter it over a larger whole array as array comprehensions work on element level.
- **More intermediate arrays.** As an array comprehension does not allow the expression of loop carried dependencies, extra intermediate arrays often need to be created.

The consequence of this lack of expressiveness of purely functional code is a high instruction count as well as a high storage use, as we shall see in the Results and Analysis section.

3.3 I-structure Implementation

An I-structure can be created empty somewhere in the code, and partly or completely filled throughout the program. An I-structure allows array elements to be defined by array element assignments, but still retains determinacy by allowing each element to be defined at most once. The I-structure implementation of this benchmark is the closest to the problem definition. Also almost nowhere were we forced to over-specify intermediate array details introduced by the purely functional style. This makes the I-structure approach in certain circumstances more declarative than the purely functional approach [3]. Array elements are now defined in loop constructs and nothing prevents us from defining more than one element in one loop body, thus avoiding the sharing problem of array comprehensions. In I-stucture code we can use loop carried dependencies to avoid the creation of unnecessary intermediate structures. As we shall see, this makes I-structure codes more efficient in instruction counts as well as space usage.

3.4 M-structure Implementation

The problem with the I-structure implementation still is that, when performing a 1-D FFT on a sub-array of the 3-D object, we need two versions of the object: the one that is read, and the one that is written. When inspecting the FFT algorithm, it is clear that exactly the same elements that are being read, need to be rewritten. An imperative algorithm would use only one data structure. M-structures allow us to do the same thing in a declarative context. It should be noted that M-structures are not needed here for expressiveness reasons: the algorithm can be expressed very naturally using I-structures. The only reason to go to an M-structure implementation of FT is space efficiency.

As was mentioned in the introduction, M-structures allow elements to be "put" into an empty location: $A![i] = expression$. A *put* cannot overwrite a value, i.e. a *put* to a full location will result in a run-time error. Elements can be extracted, i.e. read and emptied, by a "get" operation: $x = A![i]$. A third operation $x = A!![i]$ reads (or extracts and puts back) an M-structure element. A fourth operation $A!![i] = x$ replaces (extracts and puts a new value) an element of an M-structure. Notice that the operations with one ! change the full/empty state of the elements, whereas the operations with the double !! take a full element and leave it full.

M-structures in a parallel model of computation, such as the Id model of computation, give rise to non-determinism: if a number of threads try to get an M-structure element, only one can have it, and which one that will be depends on the arrival time of the particular get operation.

We found that there are two styles of M-structure programming. The *imperative* style uses reads and replaces to ensure that M-structure elements are always full, and provides explicit synchronization using barriers (notation - - -). This style mimics imperative programming closely. Of the M-structure implementations of the FT benchmark, this is the easiest to write, simply because use of replace operations on M-arrays can closely imitate imperative programming style. Use of barriers for explicit synchronization can also emulate close to sequential programming style, extending the ease of programming (to people who are more familiar with sequential thought process). This style gives us the maximum space efficiency we are after. However, the use of barriers combined with the reads and replaces, which are more expensive than gets and puts, makes this style almost as inefficient as the purely functional style!

The second style of M-structure programming, the *data driven style*, uses *puts* and *gets* for both communication and synchronization and avoids barriers as much as possible. Now the *puts* and *gets* need to be perfectly balanced, which makes this style much harder and more error prone. However, inspecting the behavior of FFT algorithms, it is clear that the butterfly data dependences in the recombination phases allow extracting two elements, performing some shared computation on these and putting two elements back in exactly the same place. Therefore, *puts* and *gets* without extra barriers work for FFTs.

3.5 Code Example

We consider the function that performs $n2$ one dimensional FFTs each of size $n1$ from an array which is $n1 \times n2$ long. This essentially requires partitioning the input vector in slices, performing a 1-D FFT on each such slice and gluing the resulting together to obtain the resulting array. An I-structure implementation of the function is as follows:

```
def cffts IS  n1 n2 x ro = {
stride = n1*n2; size = n1*2; typeof v = I_vector(F); v= I_vector(1, 2*n1*n2);
  {for j<-1 to n2 do
       y = slice j n1 stride x;  z = fft y ro IS;
       {for i<-1 to n1 do
            v[(j-1)*n1 +i] = z[i];
            v[(j-1)*n1 +i +stride] = z[n1+i];  } }
in v};
%            1-D FFT on a slice:   I-Structure Code
```

In the above code the function *slice* copies a slice of values from x, and the vector *ro* contains roots of unity. When the function is implemented in imperative M-structure style, we need to sequentialize the problem using explicit barriers before every stage, because the arrays y and z will be reused. The function *read_slice* reads a slice of values from x and updates y with these values. The function *fft* updates z. Again, *ro* contains roots of unity.

```
def cffts IS  n1 n2 x ro = {
typeof x = M_vector(F); typeof ro = I_vector(F);
stride = n1*n2; size = n1*2;
z = {M_array (1,size) of |   [j] = 0.0 || j<- 1 to size};
y = {M_array (1,size) of |   [j] = 0.0 || j<- 1 to size};
---
  {for j<-1 to n2 do
      _ = read_slice j n1 stride x y;
      ---
      _ = fft y ro IS z;
      ---
      {for i<-1 to n1 do
         x!![(j-1)*n1 +i] = z!![i];
         x!![(j-1)*n1 +i +stride] = z!![n1+i];  } }
in x};
%           1-D FFT on a slice:  Imperative M-Structure Code
```

When writing the same code in a data-driven M-structure style, we get rid off all the explicit barriers except one. Lack of this barrier causes writing on the same location of x (within the i loop), before it is extracted (by function *get_y*). The function *get_slice* extracts values out of x and puts them in y. The function fft extracts the values out of y and puts them in z, which is emptied again in the code that rewrites the slice of x that was emptied by *get_slice*.

```
def cffts IS  n1 n2 x ro = {
typeof x = M_vector(F); typeof ro = I_vector(F);
stride = n1*n2; size = n1*2;
z = 1d_X_array(1,size); y = 1d_X_array(1,size);
  {for j<-1 to n2 do
% fn get_y fills y and empties a section of x
      _ = get_y j n1 stride x y;
      _ = fft y ro IS z;
      ---
      {for i<-1 to n1 do
         x![(j-1)*n1 +i] = z![i];
         x![(j-1)*n1 +i +stride] = z![n1+i];  } }
in x};
%           1-D FFT on a slice:  Data-driven M-Structure Code
```

Building an array out of a variable number of variable sized sub-arrays turns out to be quite hard to implement in the purely functional style. The problem here is that the output value of one element of a slice is not known independently, as for one input slice, all the elements of the resulting slice are evaluated. It would be too expensive to evaluate a whole slice of elements just to get the value of one element. To get around this problem, we create an intermediate vector of vectors, each vector representing a slice.

```
def cffts IS  n1 n2 x ro = {
stride = n1*n2; size = n1*2; length = 2*stride;
typeof tv = vector(vector(F));
tv = { vector (1,n2) of
       | [j] = get1d j n1 stride x || j <- 1 to n2};

  defsubst get1d j n1 stride x = {
    y = get_y j n1 stride x;
    z = fft y ro IS;
  in z};

v = { vector (1,length) of
      | [(j-1)*n1 +i] = tv[j][i] || j <- 1 to n2; i <- 1 to n1
      | [(j-1)*n1 +i +stride] = tv[j][i+n1] || j <- 1 to n2;  i <- 1 to n1};
in v};
%           1-D FFT on a slice:  Functional Code
```

4 Results and Analysis

In this section we measure instruction counts and critical path length using the Monsoon Interpreter Mint. Two time related measures are recorded: S_I, the total number of instructions executed, and the *critical path length* S_∞, the number of parallel time steps required, when the availability of an unbounded number of processors is assumed. We measure the space requirements of our codes by determining the largest possible problem size that fits on a one node Monsoon Machine.

4.1 Time Analysis

Table 1 summarizes the instruction counts (S_1) and critical path lengths S_∞ for some problem sizes. The functional code has the maximum instruction count. This is caused by the inability to share computation in array comprehensions, and by the need to create intermediate data structures, as shown in the slice example. The I-structure code has the lowest instruction count, and a critical path length close to the functional code, which indicates that the higher parallelism of the functional code is superfluous. The data-driven M-structure code requires about 20% more instructions than the I-structure code. Also, the M-structure code has a 75% to 100% longer critical path length. This is caused by the need for explicit synchronization, and by the fact that the M-structure X, after the checksum has been computed, needs to be completely emptied before it can be reused in (producer-consumer style) in a next stage. This occurs after the checksum operation. Also, the code performing the checksum needs to use the more expensive *reads* in order to simplify the emptying of the array.

4.2 Space Analysis

To reach the goal of running a 64 x 64 x 64 size problem on the one node Monsoon machine, we first need to determine how many such arrays can be stored before we run out of heap memory. Running just the random vector generator (generating U in Figure 1), we establish that the

Method	Functional		I-structure		M-structure	
	S_1	S_∞	S_1	S_∞	S_1	S_∞
4x4x4	3,241	220	1,025	230	1,257	340
8x4x4	6,403	300	1,556	300	1,902	520
8x8x4	13,729	480	2,665	480	3,211	840
8x8x8	32,303	800	4,973	800	5,877	1,600

Table 1: S_1 and S_∞ for FT benchmark (× 1000)

maximum size of a 3-D object that can be generated on our machine is 128 × 128 × 80, and that at most 4 64 × 64 × 64 arrays can exist at the same time. According to the benchmark specification (see Figure 1), *Exp* needs to be created once and remains needed throughout the program. In the first stage of the program U and V coexist. After that, only V will be needed. In the cycle for t from 1 to 6, W and X coexist. Therefore at least 4 3-D objects must coexist. In the M-structure implementations we can over-write V on U and X on W, which brings the requirement down to 3 3-D objects. From this we conclude that on a one node Monsoon, we can never run any larger than 64^3 problem.

To establish the space usage of our 1-D FFT codes, we ran these independently of the rest of the FT benchmark. The functional code without resource management allows problem sizes of up to 2^{16}, whereas with resource management it allows problem sizes of up to 2^{18}. The corresponding numbers for the I-structure implementation are 2^{17} and 2^{18}. The M-structure code does not require explicit resource management as it reuses its data structures. It allows problem sizes up to 2^{19}. The double capacity of the M-structure code is explained by the fact that it writes the resulting fft back on its input array.

The maximal FT benchmark problem sizes we could run were 16^3 for the resource managed functional code, 32^3 for the I-structure code, and 64^3 for the M-structure code. This phenomenon corroborates our discussion in the implementation section. The functional implementation creates, even when resource managed, intermediate structures. The I-structure implementation requires independent structures for input and output, whereas the M-structure implementation can write results back into the input structure.

5 Conclusion

I-Structures and M-structures are introduced in [1] and [3], respectively. Id implementations of the 1D FFT are discussed in [4]. Sisal implementations of the 1D FFT are discussed in [5]. A high performance 1D FFT algorithm in Sisal for a vector computer is presented in [6]. The contribution of this paper is the quantitative comparison of the different declarative programming styles using a realistic, well-known benchmark.

In this paper we have studied certain declarative language features and their effect on the time and space efficiency of our programs. More specifically, we have studied three declarative implementations of the NAS FT benchmark: a purely functional, an I-structure, and an M-structure implementation, all written in the programming language Id and executed on the Monsoon Interpreter and Monsoon hardware. The purely functional code provides the most parallelism, but at the cost of a high instruction count. I-structures provides the fastest implementation: the lowest instruction count and a critical path length very close to that of the functional code. However, I-structure code is less space efficient than M-structure code. Only the M-structure code allows the 64^3 problem specified in the FT benchmark to be run. Therefore, we have the ability to trade space for time between the M-structure and I-structure implementations of this benchmark.

References

[1] Arvind, Nikhil R.S., and Pingali, K.K., "I-structures: Data Structures for Parallel Computing" ACM Transactions on Programming Languages and Systems, Vol 11, No 4, October 1989, pp 589-632.

[2] Bailey, D., et. al., "The NAS Parallel Benchmarks", Report RNR-91-002 revision 2, NASA Ames Research Center, 1991.

[3] Barth, Paul S., R. S. Nikhil and Arvind, "M-structures: Extending a parallel, non-strict, functional language with state," **Proc. Functional Prog Languages and Comp Arch**, Cambridge, MA, Aug 1991.

[4] Böhm, A. P. W. and Hiromoto R.E.,"Dataflow Time and Space Complexity of FFTs", **Journal of Parallel and Distributed Computing** no. 18, pp , 1993.

[5] Bollman, D., Sanmiguel, F. and Seguel, J., "Implementing FFT's in Sisal" Proceedings of the Second Sisal Users Conference, December 1992, Lawrence Livermore National Laboratory Report CONF-9210270.

[6] Feo, J. and Cann, D., "Developing a high-performance FFT algorithm in Sisal for a vector supercomputer", Proceedings Sisal'93, October 1993, Lawrence Livermore National Laboratory Report CONF-9310206.

[7] Hicks, James, D. Chiou, B. S. Ang and Arvind, "Performance studies of Id on the Monsoon dataflow system," **Journal of Parallel and Distributed Computing** no. 18, pp 273-300, 1993.

[8] R.S. Nikhil, *Id (version 90.0)* Reference Manual. TR CSG Memo 284-1, MIT LCS 1990.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

The Plan-Do Style Compilation Technique for Eager Data Transfer in Thread-Based Execution

M. Yasugi[a], S. Matsuoka[b] and A. Yonezawa[a]

[a]Department of Information Science, Faculty of Science, the University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan 113

[b]Department of Mathematical Engineering, Faculty of Engineering, the University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan 113

Abstract: *Plan-do* compilation technique is a new, advanced compilation framework for ***eager data transfer*** on distributed-memory parallel architectures. The technique is especially effective for a recent breed of low-latency architectures by realizing a high-throughput low-latency communication scheme, ***pipelined sends***. The compilation of high-level, plan-do style code into low-level, eager data transfer code is achieved via straightforward application of a set of translation rules. Preliminary low-level benchmark results on a real parallel architecture, EM-4, exhibit good speedups.

Keyword Codes: C.4.2; D.1.3; D.3.4
Keywords: Multiprocessors; Concurrent Programming; Language Processors

1 Introduction

Distributed-Memory Massively Parallel Processors (DM-MPPs) have attractive features for scalability. There are many types of DM-MPPs; but, irrespective of their architectures, the basic abstract execution models of such machines would be those of ***asynchronously communicating (multiple) threads***. Such an abstract machine could be naively realized by adding message-passing routines to the thread-based abstract machines; the collection of those routines is usually called a message-passing library.

However, such a library-based approach prevents us from exploiting advanced implementation techniques for communication; such techniques can be exploited only by compilation-based approaches. They include, for example, a technique of compiling message interpretations, known as ***active messages***[6], in which a message omits the runtime tags for the message format and carries the address of a compiled message handler. There are further advanced implementation techniques enabled by compiling communications.

In this paper, we concentrate on an implementation technique, ***eager data transfer***, and describe its compilation framework. The idea of ***eager data transfer*** originates from the dataflow execution model, but we employ it in the context of thread-based execution models. Eager data transfer can be (re)stated as "eagerly sending a datum to the location where the datum will be used," which improves both throughput and latency by reducing local buffering of messages on both sender and receiver nodes. Of course, there is a tradeoff for the excessive use of eager data transfer; it increases the number of messages and matchings of the messages, but we demonstrate the effectiveness when employed with compiled *pipelined sends*, via performance measurements on a fine-grained hybrid parallel architecture EM-4[4, 3].

There are many different levels of abstraction, even for the same general notion of 'directly communicating threads.' For example, Concurrent Object-Oriented Programming (COOP) languages, such as ABCL[10], provide high-level abstraction, while distributed memory machines themselves provide the lowest-level of abstraction. In this paper, we regard the compilation process as shifting down the abstraction levels within the directly communicating threads model, and discuss the necessary compiler supports.

In order to realize eager data transfer, dataflow information is indispensable. The compiler must provide additional dataflow information at every level of the abstraction; i.e., every intermediate representation must embody dataflow information to maintain modularity (clear separation of layers), even at the levels where optimization techniques based on dataflow information are not applied. This requirement causes a difficulty in combining several compilation phases into one pass. The ***plan-do style*** intermediate code solves this problem by representing the combination of control-flow information and dataflow information in a stream form. Furthermore, it has an additional benefit of serving as a convenient intermediate representation when pipelined parallelization of the compilation phases themselves is desired.

2 Eager Data Transfer

In order to clarify our motivation, this section presents several examples of eager data transfer, which can be used at various abstraction levels.

2.1 A Fine-Grained Example — Pipelined Sends

A (remote) message passing implementation scheme which overlaps computation and communication, *a pipelined send*, has been proposed for ABCL/EM-4[8, 9] (a COOP language system ABCL on EM-4). This scheme overlaps the evaluation of message arguments and their sending in a pipelined way; it improves both throughput and latency by eliminating local buffering of messages on both sender and receiver nodes. The scheme is realized by (1) remote code invocation, (2) remote write, and (3) FIFOness of network, which are efficiently supported by EM-4.

The packet exchange protocol for the pipelined message passing proceeds as follows (See Figure 1): (1) The sender reserves a message box on the receiver node, by invoking a proper code block to return the allocated message box address. (2) The sender evaluates the arguments of the message. As each argument is evaluated, its packet is eagerly sent to the proper location in the message box on the receiver node. The evaluation and sending are pipelined in a fine-grained manner. (3) The message box address is sent to the receiver. It may enqueue the message until it can process the message.

Although packet transmissions to reserve a message box may result in a round-trip latency, multi-threading at the sender node effectively hides the latency. Moreover, the reservation provides us the following significant advantages: (1) It allows us to manage the message queue efficiently at the receiver node, just by pointer manipulation rather than copying all the contents of the message. (2) The knowledge of the target message box address makes pipelined send possible, which reduces (a) *the communication latency* by eager data transfer before the complete evaluation of the message and (b) *the cost of*

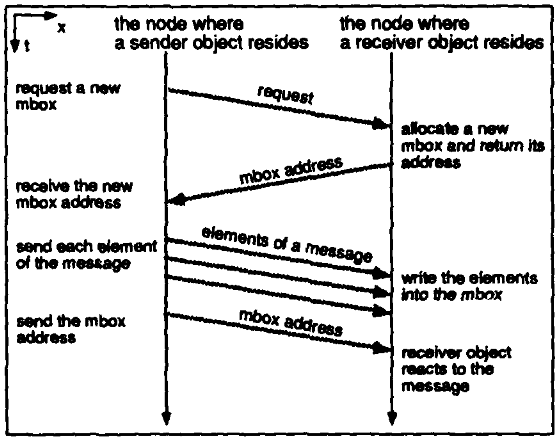

x
t
the node where
a sender object resides
the node where
a receiver object resides
request a new mbox
request
allocate a new mbox and return its address
mbox address
receive the new mbox address
send each element of the message
elements of a message
write the elements into the mbox
send the mbox address
mbox address
receiver object reacts to the message

Figure 1: Packet Exchange for a Pipelined Request Message Send

local memory access for local buffering at the sender node. Moreover, pipelined sends can be interruptible, namely, multiple sends can be interleaved with one another.

2.2 Coarser-Grained Examples

Eager data transfer can also be exploited at coarser levels of granularity:

```
[Obj1 <= [:Msg1 [:cons 1 [:cons 2 [:nil]]]]]
```

This program fragment in a COOP language, ABCL, denotes a request message send to Obj1, whose message consists of a message tag :Msg1 and a list of integers 1 and 2. Suppose that this is a remote message send and *cons cells* are used to implement the list. We would like the compiler to generate the code for *remote cons* at the node where the Obj1 resides, rather than *local cons* plus *remote copy*.

This kind of eager data transfer is not limited to COOP languages. The corresponding example in a (thread-based) functional language would be as follows:

```
(Fun1 [:cons 1 [:cons 2 [:nil]]])
```

Here, we would like the compiler to generate the code for *remote cons* at the node where the function invocation frame for Fun1 resides.

3 Compilation Framework for Eager Data Transfer

The high-level code should contain certain amount of dataflow information to realize eager data transfer at the lower-level. The dataflow information is actually needed for code conversion (generating lower-level code from higher-level code), rather than for mere code optimization. The information includes (when fixing a value v): (1) whether or not v is used eventually, (2) which thread (node) will use v, (3) to which location on memory v will be finally stored, etc. Both (2) and (3) are especially indispensable for realizing the pipelined sends described in Section 2.1.

3.1 Inlining Dataflow Information in Plan-Do Style

The *plan-do style* intermediate code satisfies the [illegible] described above and also provides additional benefits for the compilation process itself. This subsection briefly describes the plan-do style code; the important effect of using the plan-do style (i.e., realizing eager data transfer) is demonstrated when the plan-do style code is interpreted by the lower-level part of the compiler, which is presented in Section 3.2.

The basic idea behind plan-do style is that (1) the ***plan-part*** (i.e., plan declaration part) provides dataflow information, and (2) the *do-part* (i.e., plan execution part) provides control-flow information. The benefits of the plan-do style are: (1) it takes a stream form instead of an arbitrary graph structure, and (2) interleaving of plan declaration and plan execution is possible. Thus, it allows combining of several compilation phases into one pass, or alternatively, pipelined parallelization of the compilation phases.

There are compilation targets for which the plan-do style code is specially suitable, which are the *expressions* representing natural dataflow information; using the plan-do style code makes it possible to extract and represent the information without costly flow analysis. The plan-do style code embodies the semantics of the source program in terms of sequential execution of a single thread of control, and also provides dataflow information without loss of machine independence of high-level code.

Let us take a simple message send example:

```
[obj1 <= [(+ i 1) (+ j 2)]]
```

This program fragment in ABCL denotes a request message passing to object `obj1`, whose message is a tuple of an integer i+1 and an integer j+2. This source program is converted into the following plan-do style code via the high-level parse tree:

```
(newplan (p1 d1 d2) (request-send))
(throw obj1 d1)
(newplan (p2 d3 d4) (make-tuple d2))
(newplan (p3 d5 d6) (arith3-+ d3))
(throw i d5)
(throw 1 d6)
(do p3)
(newplan (p4 d7 d8) (arith3-+ d4))
(throw j d7)
(throw 2 d8)
(do p4 p2 p1)
```

`(newplan (p1 d1 d2) (request-send))` declares *plan* p1 together with ***destinations*** d1 and d2. By 'destination,' we mean an argument of the plan (which corresponds to a dataflow arc to the plan node in terms of dataflow model). The plan is to send a ***request message*** to a target object. The target object's name is given by 'throwing' the name to d1. By 'throwing,' we mean that a value is passed as an argument for a plan. `(throw obj1 d1)` throws the value of `obj1` to d1. `(do p1)` carries out the plan p1. Similarly, the actual parameter of the request message is provided by throwing the obtained value (i.e., a tuple consisting of i+1 and j+2) to d2 in a subsequent execution of another plan.

Accordingly, `(newplan (p2 d3 d4) (make-tuple d2))` declares plan p2 to (1) make a tuple of values thrown to d3 and d4 then (2) throw the tuple to d2. `(newplan (p3 d5 d6) (arith3-+ d3))` declares plan p3 to (1) add values thrown to d5 and d6 then (2) throw their sum to d3.

The translation of the parse tree into the plan-do style code is straightforward: When the (recursive) translation function reaches a certain tree node, before the translation of the subtrees, it declares the plan for the current tree node, and after the translation of the subtrees, it 'executes' the plan.

The standard (trivial) interpretation of the plan-do style code (which is not employed for eager data transfer) is as follows: (1) A plan declaration is interpreted just as declaration and no actions are performed. A destination is interpreted as a temporary variable. (2) A 'throw' is interpreted as an assignment of the value to the corresponding temporary variable. (3) A plan execution is interpreted as an actual execution of the plan. In Section 3.2, an alternative interpretation developed for eager data transfer is presented.

3.2 Interpreting Plan-Do Style Code to Generate Eager Data Transfer Code

This section describes the compilation process for realizing eager data transfer, by interpreting the high-level plan-do style code. We exemplify the process for pipelined sends described in Section 2.1 with the simple message send example in the previous section.

Figure 2 shows the translation function *Tr* from the high-level code into the lower-level code. Here, when *hcode* is high-level plan-do style code, $Tr[\![hcode]\!](penv, denv)$ returns the lower-level code in the context of the plan environment *penv* and the destination environment *denv*. *penv* is a function from a set of plan identifiers (ranged over by p) to plan definitions (p corresponds to a dataflow node in terms of the dataflow model). *denv* is a function from a set of destination identifiers (ranged over by d) to pairs of p and integers (ranged over by i). (d stands for i-th dataflow arc to p in terms of the dataflow model.) $Trnsfr[\![v_l, fl, d]\!](penv, denv)$ returns a fragment of lower-level code which transfers a value v_l to the fl-field of destination d. The results of *Trnsfr* are significantly influenced by the dataflow context of (*penv*,*denv*) in order to select the eager data transfer instructions under certain conditions. For a function f, "$f\{x \mapsto y\}$" denotes the function f' such that $Dom(f') = Dom(f)$, $f'(x) = y$ and $f'(x') = f(x')$ for all $x' \in Dom(f) - \{x\}$. By "$h :: r$," "::" denotes consing of the head h and list r, and "$[a, b, c]$" denotes the list consisting of a, b, and c. By "l_1 @ l_2," "@" denotes the concatenation of two lists l_1 and l_2.

The translation function basically translates the high-level code in the following way:

- For a plan declaration, it memorizes the plan and the destinations in environments *penv* and *denv*. Fresh lower-level variable identifiers will be introduced if necessary.
- For a plan execution, it generates code for finishing the plan, then if the result value has not been eagerly sent, it generates data transfer instructions by calling the *Trnsfr* function.
- For throwing a value to a destination, it decomposes the (possibly) nested higher-level tuple value into a set of pairs of lower-level values and the field number lists by calling *Decomp*, and generates data transfer instructions by calling *Trnsfr*.

Tr and *Trnsfr* embody the translation into eager data transfer; i.e., they realize the pipelined sends. For this purpose, *Tr* translates the high-level code in the following way:

- For request message passings, it prepares pointers for a request message box to realize a pipelined send, and it only generates instructions for pointer manipulation.
- For tuple construction, it eagerly transfers the tuple elements without buffering.
- For simple arithmetic operations, it simply prepares three temporary variables for the operation, and the result of the operation is transferred by *Trnsfr*.

$Tr\llbracket(\texttt{newplan}\ p_1\ d_1\ d_2\ (\texttt{req-send}))::r\rrbracket(penv, denv) =$
$\quad Tr\llbracket r\rrbracket(penv\{p_1 \mapsto \mathbf{req\text{-}send}(t_1,t_2,t_3)\}, denv\{d_1 \mapsto (p_1,1), d_2 \mapsto (p_1,2)\})$
$\quad (t_1,\ t_2,\ t_3$ new$)$

$Tr\llbracket(\texttt{newplan}\ p_1\ d_1\ d_2\ (\texttt{make-tuple}\ d_3))::r\rrbracket(penv, denv) =$
$\quad Tr\llbracket r\rrbracket(penv\{p_1 \mapsto \mathbf{make\text{-}tuple}(d_3)\}, denv\{d_1 \mapsto (p_1,1), d_2 \mapsto (p_1,2)\})$

$Tr\llbracket(\texttt{newplan}\ p_1\ d_1\ d_2\ (\texttt{arith3-+}\ d_3))::r\rrbracket(penv, denv) =$
$\quad Tr\llbracket r\rrbracket(penv\{p_1 \mapsto \mathbf{arith3\text{-}{+}}(d_3,(t_1,t_2,t_3))\}, denv\{d_1 \mapsto (p_1,1), d_2 \mapsto (p_1,2)\})$
$\quad (t_1,\ t_2,\ t_3$ new$)$

$Tr\llbracket(\texttt{do}\ p_1)::r\rrbracket(penv, denv) =$
$\quad (\texttt{get-read-pointer}\ t_2\ t_3)::(\texttt{req-send}\ t_3\ t_1)::Tr\llbracket r\rrbracket(penv, denv)$
$\quad$ if $penv(p_1) = \mathbf{req\text{-}send}(t_1,t_2,t_3)$

$Tr\llbracket(\texttt{do}\ p_1)::r\rrbracket(penv, denv) = Tr\llbracket r\rrbracket(penv, denv)\qquad$ if $penv(p_1) = \mathbf{make\text{-}tuple}(d_1)$

$Tr\llbracket(\texttt{do}\ p_1)::r\rrbracket(penv, denv) =$
$\quad (\texttt{arith3-+}\ t_1\ t_2\ t_3)::(Trnsfr\llbracket t_3, \mathbf{nil}, d_1\rrbracket(penv, denv) @ Tr\llbracket r\rrbracket(penv, denv))$
$\quad$ if $penv(p_1) = \mathbf{arith3\text{-}{+}}(d_1,(t_1,t_2,t_3))$

$Tr\llbracket(\texttt{throw}\ v_h\ d_1)::r\rrbracket(penv, denv) =$
$\quad (Trnsfr\llbracket v_{l1}, fl_1, d_1\rrbracket(penv, denv) @ \cdots @ Trnsfr\llbracket v_{ln}, fl_n, d_1\rrbracket(penv, denv)) @$
$\quad Tr\llbracket r\rrbracket(penv, denv)$
$\quad$ where $Decomp(v_h) = \{(v_{l1}, fl_1), \ldots, (v_{ln}, fl_n)\}$

$Trnsfr\llbracket v_l, \mathbf{nil}, d_1\rrbracket(penv, denv) = [(\texttt{assign}\ v_l\ t_1), (\texttt{get-rqst-mbox-on}\ t_1\ t_2)]$
$\quad$ if $denv(d_1) = (p_1, 1)$ and $penv(p_1) = \mathbf{req\text{-}send}(t_1,t_2,t_3)$

$Trnsfr\llbracket v_l, fl, d_1\rrbracket(penv, denv) = [(\texttt{setarg-remote}\ v_l\ t_2\ fl)]$
$\quad$ if $denv(d_1) = (p_1, 2)$ and $penv(p_1) = \mathbf{req\text{-}send}(t_1,t_2,t_3)$

$Trnsfr\llbracket v_l, fl, d_1\rrbracket(penv, denv) = Trnsfr\llbracket v_l, (i-1)::fl, d_2\rrbracket(penv, denv)$
$\quad$ if $denv(d_1) = (p_1, i)$ and $penv(p_1) = \mathbf{make\text{-}tuple}(d_2)$

$Trnsfr\llbracket v_l, \mathbf{nil}, d_1\rrbracket(penv, denv) = [(\texttt{assign}\ v_l\ t_i)]$
$\quad$ if $denv(d_1) = (p_1, i)$ and $penv(p_1) = \mathbf{arith3\text{-}{+}}(d_3,(t_1,t_2,t_3))$ (for $1 \le i \le 2$)

Figure 2: Tıanslation Function from Plan-Do Style Code into Lower-Level Code for Pipelined Sends.

Trnsfr realizes data transfer to the destination d in the following way :

- If d is for a target of request message send, the value is simply assigned to the temporary variables for the target, but, immediately after that, it allocates a message box for the pipelined send.

- If d is for an argument of a message, it generates a "remote write" instruction for the pipelined send.

- If d is for a tuple construction, it eagerly transfers the value to the more specific field of d by recursively calling *Trnsfr*.

- If d is just for an arithmetic operation, the value is simply assigned to the corresponding temporary variable.

Figure 3 shows the result of the conversion from the high-level code into lower-level code for `[obj1 <= [(+ i 1) (+ j 2)]]`. The high-level code is also shown on the left

```
Plan-Do Style Code                       Lower-Level Code
(newplan (p1 d1 d2) (request-send))
(throw obj1 d1)                         (assign obj1-0 t1)
                                        (get-rqst-mbox-on t1 t2)  [*1]

(newplan (p2 d3 d4) (make-tuple d2))
(newplan (p3 d5 d6) (arith3-+ d3))
(throw i d5)                            (assign i-0 t4)
(throw 1 d6)                            (assign 1 t5)
(do p3)                                 (arith3-+ t4 t5 t6)
                                        (setarg-remote t6 t2 (0)) [*2]

(newplan (p4 d7 d8) (arith3-+ d4))
(throw j d7)                            (assign j-0 t7)
(throw 2 d8)                            (assign 2 t8)
(do p4)                                 (arith3-+ t7 t8 t9)
                                        (setarg-remote t9 t2 (1)) [*3]

(do p2)
(do p1)                                 (get-read-pointer t2 t3)  [*4]
                                        (request-send t3 t1)      [*5]
```

[*1] allocate a message box t2 on the node of t1, [*2] pipelined remote write of i+1, [*3] pipelined remote write of j+2, [*4] complete the initialization of the message box, [*5] send the message box address.

Figure 3: Generated Lower Level Code for [obj1 <= [(+ i 1) (+ j 2)]]

hand side. Eager data transfers become manifest at (do p3) and (do p4), where the elements of the tuple are eagerly sent to the pre-allocated message box on a remote node. Note that (do p2) does nothing because the tuple elements are already sent eagerly.

4 Performance Measurements

In this section, we demonstrate the effect of eager data transfer, especially pipelined sends, via performance measurements on a COOP language system ABCL/EM-4[8, 9].

The source language of ABCL/EM-4 is ABCL/ST, a statically-typed ABCL language. An ABCL/ST compiler for EM-4 has already been developed [7]. The compiler adopts the following language hierarchy for each level of the abstraction of 'directly communicating thread': (1) the source language, (2) high-level language (embodying the semantics of the source language in terms of sequential execution of a single thread of control), (3) middle-level language (introducing pointers for the purpose of implementation), (4) low-level language (introducing the notion of data size and memory addresses), (5) I-language (for optimization based on flow analysis), and (6) assembly language. The plan-do style is mainly used for the high-level code to provide dataflow information without loss of machine independence of the high-level code.

The compiler generates assembly code for a fine-grained *hybrid* parallel architecture EM-4[4, 3], which was developed and built at Electrotechnical Laboratories. EM-4 consists of 80 processing elements and runs at 12.5 MHz clock speed, and facilitates a fine-grained communication mechanism; for example, it provides a two-words size packet-output instruction which directly sends data from registers into its fast omega network. EM-4 hardware also provides basic scheduling mechanism for multiple threads, coupled with its data-driven (packet-driven) feature. The term "hybrid" indicates the combination of

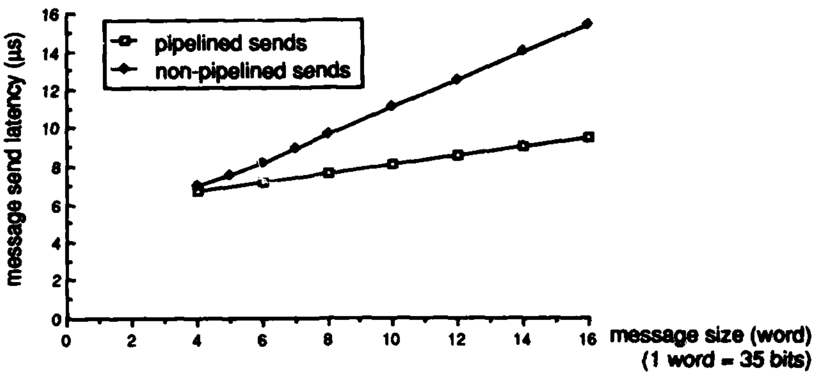


Figure 4: Latency Comparison Between Pipelined/Non-Pipelined Sends

control-flow architecture and the data-driven architecture.

In this measurement, we compare the two implementation schemes of remote mess. e passing: (1) the pipelined sends (see Section 2.1) and (2) the non-pipelined sends, in which the communication is not initiated until all the message arguments are evaluated.

In order to measure the latency of remote message sends, our benchmark program creates objects along a closed Hamilton path among the 80 nodes of EM-4, and sends messages through the path. When an object O_i is activated by a message M_i from O_{i-1}, O_i sends M_{i+1} to O_{i+1} at the adjacent node along the path.

We measured the average activation interval of adjacent objects O_i and O_{i+1} for various sizes of messages. Figure 4 shows the results of the benchmark. As we can see, the pipelined sends are always superior to the non-pipelined sends. The main reason is because the non-pipelined sends require additional memory accesses to buffer the message arguments at the sender node; this buffering is not necessary for the pipelined sends, where the evaluated value on a register is sent directly into the network.

The latency given in Figure 4 includes not only the communication latency but also the execution time of computation performed by objects. Nevertheless, the minimum latency is approximately 6.7 μs (83 clock cycles) for 4-word message size in pipelined sends, while the difference between the latency in non-pipelined sends and the latency in pipelined sends for 16-word message size is approximately 5.8 μs (73 clock cycles). These values show that the impact of pipelined sends becomes relatively significant on architectures where the remote communication is very fast.

5 Discussion

5.1 Moderate Eagerness

The idea of eager data transfer originates from the dataflow execution model. In the dataflow execution model (the *eager evaluation* model), a large number of fine-grained concurrency can be exploited, ideally resulting in perfect speed up of the execution. In practice, however, the explosion of concurrency frequently causes problems, such as the resource exhaustion and the network contention.

In the thread-based execution model, concurrency can be easily bounded by the number of threads. Moreover, each thread execution can be efficiently performed by the use of pipelines and registers. Recent approaches, such as TAM[2], attempt to take advantage of this fact by resorting to compiler-assisted, highly-efficient thread-based execution model, even for languages which have fine-grained lenient semantics.

Our approach introduces eagerness in the compilation of communication in the directly-communicating threads model to exploit the fine-grained capabilities of recent architectures. We note, however, this does not cause the same problem (explosion of concurrency) as *eager evaluation*: in our approach, the eagerness appears only in the form of eager data transfer, which does not include "fire" as in dataflow execution (which consists of eager data transfer + matching + fire). The eagerness in our approach is still well moderated by the thread-based execution model.

5.2 Scheduling

Our abstract machine model employs a simple hierarchy of scheduling, namely (1) scheduling of threads and (2) scheduling of instructions. In ABCL/EM-4, scheduling of threads is basically performed by the hardware at runtime, because EM-4 quite efficiently supports the scheduling of threads, including creation and termination of threads, with its data-driven features. Instruction scheduling within a thread is the responsibility of the compiler, and the plan-do style is employed for this purpose. But plan-do style actually plays a more important role in instruction conversion between different abstraction levels.

5.3 Combining with Other Optimization Techniques

When an optimization based on dataflow information is already required at the higher-level, generation of plan-do style code described in Section 3.1 is not necessary. In such cases, only the conversion into the lower-level code described in Section 3.2 needs to be applied for realizing eager data transfer.

For example, in array computations, [1] generates optimized code of 'directly communicating threads,' where the optimization is based on dataflow information (rather than data dependency). In non-strict computations, [5] generates optimized code of 'directly communicating threads,' by partitioning the dataflow graph. In both cases, by keeping the dataflow information in the threads, our compilation scheme described in Section 3.2 would be extended to serve as a backend which supports their communication, when compiling into further fine-grained communication is desired for architectures such as EM-4.

6 Conclusion

We have presented a novel compilation-based approach to realizing high-throughput low-latency communication for the execution models based on *directly communicating threads*. We employed a compilation-based approach rather than a library-based one, exploiting advanced implementation techniques for communication. In particular, we concentrated on *eager data transfer* and described its compilation framework. We demonstrated the effectiveness of eager data transfer when employed with compiled *pipelined sends*, via performance measurements on a fine-grained hybrid parallel architecture EM-4.

In order to realize eager data transfer, dataflow information is indispensable to code conversion. The compiler must provide additional dataflow information even at the levels where optimization techniques based on dataflow information are not applied. This requirement causes a difficulty in combining several compilation phases into one pass. Our *plan-do style* intermediate code solves this problem by representing the combination of control-flow information and data-flow information in a stream form.

Acknowledgements

We are grateful to Drs. Yoshinori Yamaguchi, Mitsuhisa Sato, Yuetsu Kodama at ETL and Dr. Shuichi Sakai at RWCP for allowing us to use EM-4 and sparing time for technical discussions. We also thank Prof. Kei Hiraki at the Univ. of Tokyo for his helpful comments. Hidehiko Masuhara's comments on an early draft are appreciated. Masahiro Yasugi is supported in part by a JSPS postdoctoral Fellowship.

References

[1] S. P. Amarasinghe and M. S. Lam, "Communication Optimization and Code Generation for Distributed Memory Machines," *Proc. of PLDI'93*, pp. 126–138, 1993.

[2] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek, "Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine," *Proc. of ASPLOS IV*, pp. 166–175, April 1991.

[3] Y. Kodama, S. Sakai, and Y. Yamaguchi, "A Prototype of a Highly Parallel Dataflow Machine EM-4 and its Preliminary Evaluation," *Proc. of InfoJapan '90*, pp. 291–298, 1990.

[4] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba, "An Architecture of a Dataflow Single Chip Processor," *Proc. of ISCA'89*, pp. 46–53, June 1989.

[5] K. R. Traub, D. E. Culler, and K. E. Schauser, "Global Analysis for Partitioning Non-Strict Programs into Sequential Threads," *Proc. of LFP'92*, June 1992.

[6] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," *Proc. of ISCA'92*, pp. 256–266, May 1992.

[7] M. Yasugi, "A Concurrent Object-Oriented Programming Language System for Highly Parallel Data-Driven Computers and its Applications," Technical Report 94-7e, Department of Information Science, University of Tokyo, Apr. 1994. (Doctoral Thesis).

[8] M. Yasugi, S. Matsuoka, and A. Yonezawa, "ABCL/onEM-4: A New Software/Hardware Architecture for Object-Oriented Concurrent Computing on an Extended Dataflow Supercomputer," *Proc. of ICS'92*, pp. 93–103, July 1992.

[9] A. Yonezawa, S. Matsuoka, M. Yasugi, and K. Taura, Implementing Concurrent Object-Oriented Languages on Multicomputers, *IEEE Parallel & Distributed Technology*, Vol. 1(2), pp. 49–61, May 1993.

[10] A. Yonezawa, editor, *ABCL: An Object-Oriented Concurrent System — Theory, Language, Programming, Implementation and Application*, The MIT Press, 1990.

PART III

MEMORY AND CACHE ISSUES

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

A Compiler-assisted Scheme for Adaptive Cache Coherence Enforcement

Trung N. Nguyen,[*] Farnaz Mounes-Toussi,[†] David J. Lilja,[†] Zhiyuan Li[*]

[*]Department of Computer Science, University of Minnesota, Minneapolis, MN 55455
[†]Department of Electrical Engineering, University of Minnesota, Minneapolis, MN 55455

Abstract: Cache coherence mechanisms in shared-memory multiprocessors typically use either updating or invalidating to prevent access to stale data, but neither enforcement strategy is the best choice for all programs. We present a compile-time optimization that uses the look-ahead capability of the compiler to select updating, invalidating, or neither for each write reference in a program to thereby produce the best overall memory performance. We implement this optimization in the Parafrase-2 compiler for memory references to scalar variables and use trace-driven simulations to compare the performance of this compiler-assisted adaptive coherence enforcement to hardware-only mechanisms. We find that this compiler optimization can produce miss ratios comparable to those produced by an updating-only mechanism while frequently reducing the total network traffic to below that produced by any of the hardware-only mechanisms.

Keyword Codes: B.3.3; C.1.2; C.5.1
Keywords: Performance Analyis and Design Aids; Multiple Data Stream Architectures (Multiprocessors); Large and Medium ("Mainframe") Computers

1 Introduction

In a shared-memory multiprocessor with private data caches, each processor may have a copy of the same memory location resident in its cache. To ensure correct program execution in such an environment, some coherence mechanism, such as *invalidating* or *updating*, is required to prevent access to stale data. An invalidating mechanism maintains coherence by requesting exclusive access to a shared memory location before a write operation. This exclusive access request causes all other cached copies of the memory location to be marked as *invalid* in each processor's cache. Any references to an invalid copy cause a cache miss and thereby force the processor to access the most recent copy of the memory location from the globally shared memory. The updating strategy, on the other hand, ensures coherence by distributing the newly written value of a cached memory location to all other caches with a valid copy every time the location is written.

Several studies [11, 1, 19, 13] have shown that, in general, neither an updating mechanism nor an invalidating mechanism alone can produce the best overall performance.

This work was supported in part by National Science Foundation grants CCR-9209458 and CCR-9210913.

For some programs, updating will produce the lowest average memory delay, while for other programs, invalidating produces the best result. In fact, updating may produce better performance during some phases of a program's execution while invalidating may be preferable in other sections of the same program.

Hybrid adaptive schemes attempt to combine the best aspects of both strategies. Most of these schemes, supported by different types of hardware mechanisms, switch between updating and invalidating at run-time based on the program's memory referencing behavior [3, 15, 21, 12, 5, 6]. Some researchers acknowledge the potential of using the compiler to choose between updating and invalidating [16, 2], but they have not developed algorithms for performing this switching. We have recently proposed an adaptive scheme [18] that utilizes compile-time information to identify write references that require coherence enforcement and to select updating or invalidating for each of these references.

In this paper we present specific compiler algorithms to perform the type of coherence enforcement marking mentioned above. We implement these algorithms in the Parafrase-2 parallelizing compiler [20]. Trace-driven simulations are used to determine the performance improvement of this compiler-assisted scheme compared to nonadaptive schemes and to adaptive schemes that use only run-time information. The ideas in this paper apply to both scalar and array references. However, due to the lack of precise array data flow analysis in the compiler used for the experiments, we focus on scalar references only. We always use write invalidation for array references. Currently, we are developing a compiler that will include exact array dependence analysis allowing us to handle array references in our future work.

1.1 Program Model and System Assumptions

In this work, we consider the parallel execution of programs in the form of DOALL loops. A DOALL loop cannot have any data dependences between different iterations. In addition, it will terminate only when all iterations are completed. Due to the complexity of nested DOALL loops, we consider only singly-nested DOALL loops. If several nested loops are parallelizable, we make the outermost one a DOALL loop. Processors may be assigned at the entry and the exit of a DOALL loop, which we will call a *boundary*. All synchronizations occur at these boundary points. We assume that the hardware will provide the necessary support for synchronization.

For our simulations, we assume a system of multiprocessors that are connected to the shared memory via a packet-switched multistage interconnection network. Each processor has a private data cache. Sequential regions of a program (i.e. those that are not DOALL loops) are always executed by the same processor. The system has a directory to monitor which processors have a valid copy of each block. There are several variations of directories [8, 4, 9, 14], any of which can be used with this compiler optimization. Here we adopt a directory structure similar to Censier and Feautrier's [8] in which each memory module is associated with its own directory. We also assume that each processor may issue three types of writes: *wrt-up*, *wrt-inv*, and *wrt-only*. The *wrt-up* and *wrt-inv* instructions are used to write a shared-writable block and invoke either an updating or an invalidating coherence enforcement strategy, respectively. The *wrt-only* instruction is used for references which cannot cause incoherence. The compiler's task is to identify how each write reference should be marked, and then generate the corresponding instruction to perform the necessary type of write operation.

```
(1)   y = a + b
(2)   y = y * y          → last write to y
(3)   IF (x > 5) THEN
(4)     y = x + y        → last write to y
(5)   ELSE
(6)     x = 0            → last write to x   → coherence write to x
(7)   ENDIF
      —< boundary >—
(8)   DOALL i=1, 10
(9)     A[i] = x + i
(10)  END DOALL
      —< boundary >—
(11)  ... = x + y
```

Figure 1: An example of a program segment showing how the writes are marked.

2 Compiler Analysis

The algorithm in this work applies to individual routines separately, although it can also be extended to an interprocedural analysis. The analysis consists of two primary components: 1) marking those writes that may require coherence enforcement by calling functions *find_last_writes* and *identify_write_only*; and 2) choosing the appropriate coherence action (updating or invalidating) for these writes by calling functions *estimate_degree_of_sharing* and *determine_write_type*.

2.1 Marking last writes and recognizing *wrt-only*

Not all write references need to incur coherence enforcement actions. For example, if the value written by a processor will be overwritten by the same processor, and thus will not be used by any other processors, then the coherence enforcement can be delayed until the last write which can potentially be followed by a read of that block by a different processor. The first part of our analysis is to recognize these last writes. The remaining writes cannot produce values used by other processors and, therefore, cannot create incoherence. These writes can be safely marked as *wrt-only*.

The function *find_last_writes* identifies the writes that can reach the end of a parallel or sequential region. These last writes can potentially cause cache incoherence. First, the function finds all boundaries in a routine. Next, for each boundary node, B, a control flow subgraph (CFS), $F(B)$, is constructed. $F(B)$ contains all program segments that are reachable from B without crossing another boundary. After the CFS is built, the function *mark_last_write* is called to mark the last writes. This function is a simple variation of the iterative algorithm commonly used to find the reaching definitions. The *reach* sets of all leaf nodes in the CFS contain the last writes. Any other writes are marked as *wrt-only*. Figure 1 shows an example of how the last writes are marked. In this segment of code, the write to y on line 4 and the write to x on line 6 will be marked as last writes since they are the last writes before the boundary is crossed. We must also mark the write to y on line 2 as a last write since the path taken at the if-statement cannot be determined at compile-time. The write to y on line 1 is not a last write so it can be marked *wrt-only*.

Not all of the last writes marked by the function *find_last_writes* will cause cache incoherence, however. Some of these writes may never reach a read by a different processor. The ones that will not be read by another processor are marked as *wrt-only* in function *identify_write_only*. The remaining last writes are called *coherence writes*. In our example in Figure 1, if we assume that the analyzed routine ends on line 11, the values of y written on lines 2 and 4 will never be used by a different processor and, hence, those writes are not coherence writes and will also be marked as *wrt-only*. We determine that a write can reach a read by another processor by computing upward exposed uses [10, 17]. Since this is a familiar subject, we omit the details.

2.2 Determining coherence actions for the coherence writes

Any writes that are not marked as *wrt-only* in the previous step are writes which may cause cache coherence problems. To choose between write-update and write-invalidate, the compiler must consider the cost of using each strategy for each write reference. In this paper, we define the cost to be the total network traffic. For instance, for a given write reference, the cost of using invalidating depends on the network traffic produced by the invalidation messages, plus the traffic produced by the cache misses generated by those processors which reference the block after the write. The cost of using updating, however, depends on the number of processors that have a copy of the block cached when the write occurs since each of these processors must receive an update message.

To approximate the required numbe.· of update or invalidate messages we estimate the *degree of sharing*, which is the number of processors that have a cached copy of the referenced block, before and after the write. For updating, the degree of sharing before the write is a good estimate of the number of update messages that must be sent. This sharing then can be used to estimate the cost of using updating. The degree of sharing before the write also is used to estimate the cost of sending the invalidation messages. In addition, the number of misses expected to be generated by processors reading the block after the write is estimated by the degree of sharing after the write. The cost of invalidating is then the sum of the cost of sending the invalidation messages and the read misses. After estimating the cost of each strategy, the compiler marks the write reference as *wrt-up* if the cost of updating is lower than invalidating. Otherwise, the write is marked as *wrt-inv*. Note that the actual cost of these operations is a function of the implementation details of the specific multiprocessor.

For scalar variables, the cost estimation can be simplified even further. The degree of sharing before or after a scalar write can be one of only three cases:

- 0, if the variable is not referenced before or after the write;
- 1, if the variable is referenced only in sequential regions before or after the write;
- p, if the variable is referenced in one or more parallel regions, where p is the number of processors.

We use p to denote the degree of sharing of scalar variables referenced in a parallel region since in a parallel region, a referenced scalar is usually shared by all or most of the processors. If a scalar is written in a sequential region and later is read in a parallel region, we say the degree of sharing after the write is p. In this case, the cost of the read misses after the write plus the invalidating cost during the write would exceed the cost of updating, no matter how many processors have a copy of the variable before the write. This is because reading copies of the missing data incurs at least the same amount of network traffic as sending copies of the data. In addition, a write-invalidate requires some invalidating traffic. Therefore, in this case, we mark the write as *wrt-up*.

If the degree of sharing after a write is 0 or 1, we cannot be sure of the exact degree of sharing without interprocedural analysis. We do know, however, that a degree of 1 means that the writing processor reads the variable again and that a degree of 0 means that the writing processor will not reference the variable again within this routine. In the latter case, the variable will most likely be referenced again outside the routine by all p processors. Hence, we can mark those writes with a degree of sharing of 0 after the writes as *wrt-up* (again because the read miss and invalidating cost wold outweigh the updating cost). Similarly, if the degree of sharing after a write is 1, the variable will likely not be referenced by other processors. Therefore, we can mark such writes as *wrt-inv*. Hence, for scalar variables, the calculation of the costs is simplified to determining the degree of sharing after the write.

We denote the degree of sharing of each variable by a 2-tuple $< var, count^{var} >$, where $count^{var}$ is the degree of sharing of the variable var. We estimate the degree of future sharing by using an *accumulating operator* defined as follows.

Definition of the accumulating operator: Let VAR be the set of all scalar variables in the analyzed routine. Suppose s_i is a set of variables and their estimated degrees of sharing such that $s_i = \{< var, count_i^{var} >|_{var \in \mathrm{VAR}}\}$ where $i = 1, 2, ..., k$. Then,

$$\Theta(s_1, s_2, ..., s_k) = \{< var, max(count_1^{var}, count_2^{var}, ..., count_k^{var}) >|_{var \in \mathrm{VAR}}\}$$

The accumulating operator, Θ, is used to accumulate the degree of future sharing through an execution path which does not contain any branching points and to estimate the degree of future sharing at any branching point of several potential execution paths. When accumulating the degree of sharing through an execution path, we take the maximum degree of sharing of each scalar variable since this number represents the number of processors that may have a copy of the variable during the execution of this path. Likewise, at the branching point of several potential execution paths, we select the maximum degree of sharing of a variable in these paths to represent the potential degree of future sharing of the variable. We make this selection because it represents the best choice in many frequent cases. For example, since the number of iterations of a DOALL loop typically will be far greater than the number of processors, a processor will execute many iterations. If a scalar is referenced in any potential execution path, then by executing several iterations, a processor will very likely reference the scalar at run time. If a scalar is read in a serial loop, its reference in any potential execution path makes it even more likely to be referenced at run time.

To estimate the degree of sharing for all scalar variables, we propagate two sets, *RA_in* and *RA_out* in the backward order and we iterate until these two sets become stable. The function *estimate_degree_of_sharing* estimates such sharing information. To make the explanation of this function clearer, the sets involved in the estimation of the degree of sharing are described here:

- *rb* - a set of $< var, count^{var} >$ tuples for a basic block. If there is no coherence write to *var* in the basic block, *b*, then

$$count^{var} = \begin{cases} 0, & \text{if } var \text{ is not referenced in } b; \\ 1, & \text{if } var \text{ is referenced in } b \text{ in a sequential region;} \\ p, & \text{otherwise.} \end{cases}$$

If there is a coherence write to *var*, then

$$count^{var} = \begin{cases} 0, & \text{if } var \text{ is not referenced before the write in } b; \\ 1, & \text{if } var \text{ is referenced before the write in } b \text{ in a sequential region;} \\ p, & \text{otherwise.} \end{cases}$$

- RA_in - the set of $< var, count^{var} >$ tuples accumulated at the bottom of a basic block during the backward propagation.
- RA_out - the set of $< var, count^{var} >$ tuples that will be propagated outside a basic block.

The set *RA_in* of each flowgraph node contains the degree of sharing information given by the successors of the node. When a node has more than one successor (i.e. a branching point), we estimate the degree of future sharing by using the accumulating operator Θ defined earlier. To obtain the *RA_in* set, we need the set *RA_out*, which contains the degree of sharing information for variables which are not killed by the coherence writes in the node. We must also include the degree of sharing information local to a node, the set *rb*, that can be live outside the node in the *RA_out* set. When accumulating the degree of sharing information, we again use the Θ operator.

After obtaining the necessary degree of sharing information, we can determine the cost of updating and invalidating for a coherence write reference. As discussed earlier in this section, the costs for scalar writes are simplified to examining the degree of sharing after the writes. The function ***determine_write_type*** uses the degree of sharing obtained from function *estimate_degree_of_sharing* to mark the coherence writes as *wrt-up* or *wrt-inv*. In the example in Figure 1, for instance, the coherence write to x on line 6 will have an *RA* count (degree of sharing after the write) of p. Note that after this coherence write, x is referenced in the DOALL loop on line 9 and, therefore, will have a degree of sharing of p. It is referenced again in the serial section on line 11 with a degree of sharing of 1. When estimating the degree of sharing, the accumulating operator yields p as the degree of sharing of x after the write on line 6. Since the *RA* count for x at this point is p, the coherence write to x will be marked as *wrt-up*.

3 Simulation Methodology and Results

We compare the relative effectiveness of invalidating-only [8], updating-only, a dynamic adaptive scheme that uses only run-time information, and our compiler-assisted adaptive scheme with respect to the *miss* traffic and the *coherence* traffic on several of the Perfect Club benchmark programs [7]. The miss traffic is the miss ratio times the number of bytes required to service a miss. The coherence traffic includes the invalidating, updating, write-back, and write-through traffic necessary to maintain coherence among the caches and the shared memory. The updating scheme uses write-update instead of write-invalidate to enforce coherence. The dynamic adaptive scheme enforces coherence similar to the updating protocol as long as the number of consecutive writes to the same block with no intervening cache misses (read or write) is less than a predefined threshold value. When the number of consecutive writes to a block reaches this threshold value, all cached copies of the block, except for the writing processor's copy, are invalidated. This dynamic adaptive scheme is our directory-based variation of the bus-based adaptive schemes [3, 15]. A detailed discussion of these protocols can be found in [18].

Separate forward and reverse networks with 32-bit data paths connect the 16 processors to the shared memory. Each packet requires a minimum of two words: one word for the source and destination module numbers plus a code for the operation type, and another word for the actual memory block address. One or more additional words are used for fetching and writing data, or for sending data updates to other processors.

The Parafrase-2 compiler is instrumented to generate a trace of the memory addresses produced by each of the 16 processors. These traces are randomly interleaved into a single trace following the *fork-join* parallelism model. To simulate the compiler-assisted adaptive scheme, the Parafrase-2 compiler marks the memory references using the marking algorithm described in Section 2. The marked trace then drives a cache simulator to determine the miss and coherence traffic.

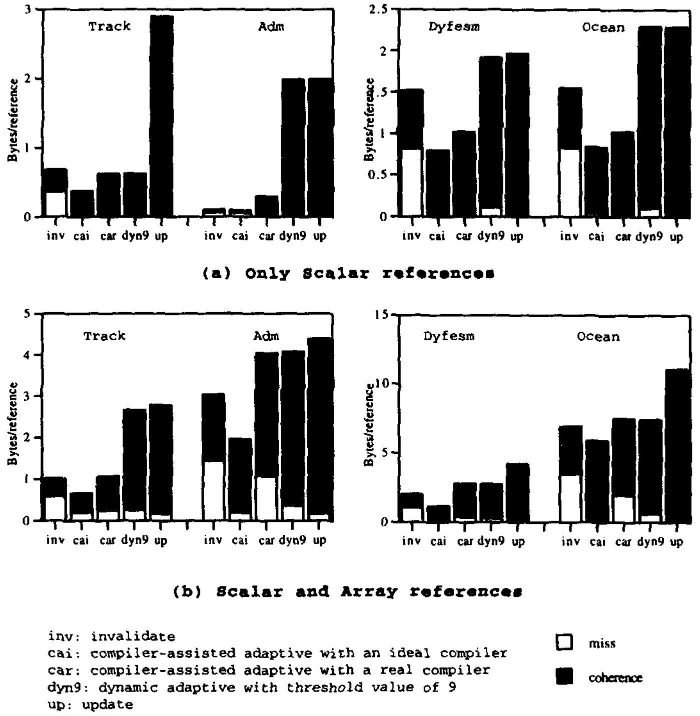
To focus on the performance of only the compiler optimizations, an infinite size, fully associative cache with a block size of one-word is used in each of the processors. To evaluate the performance of this scheme in a more realistic environment, we varied the block size of a 2 Kword cache from 1 to 2 and 4 words using three different block placement policies: direct-mapped, 4-way set-associative, and fully associative. However, since the simulations with different placement policies showed no significant difference, only the simulation results of the set-associative cache is included in this paper. All instruction references are ignored since they can never cause any coherence problems. Since we are interested in the effect of coherence enforcement only on data references, synchronization variables are not considered in these simulations.

Figure 2(a) shows both the miss and the coherence network traffic for scalar references only. The *ideal* compiler result uses the traces to simulate a compiler capable of performing perfect memory address disambiguation, perfect interprocedural analysis and perfect branch prediction. The *real* (Parafrase-2) compiler, on the other hand, must be conservative when a branch, an array reference, or a procedure call is encountered. While the *ideal* case optimizes both array and scalar references, the *real* compiler optimizes only the scalar references.

As shown in Figure 2(a), the compiler-assisted scheme is quite effective in reducing the total network traffic. In fact, the miss traffic and, therefore, the miss ratio, generated by the *ideal* compiler and the *real* compiler is up to 99 percent lower than the invalidating scheme and the dynamic adaptive scheme. This improvement in misses is due to the frequent selection of an updating protocol at compile-time. While the miss traffic of the compiler-assisted scheme is almost the same as the miss traffic generated by the updating scheme, the coherence traffic is 49 to 98 percent lower than the updating scheme. It also is up to 98 percent lower than the dynamic adaptive scheme. The coherence traffic of the three hardware-only coherence schemes is mainly due to the unnecessary block updates and invalidates. A comparison of the *ideal* and the *real* compilers indicates that the analysis described in Section 2 is marking most of the write references as well as the *ideal* compiler can mark them. The performance difference between the two compilers is primarily due to procedure calls and branches that are ignored in the ideal case.

While the previous discussion considered only scalar references, Figure 2(b) shows the simulation results with both array and scalar references. As shown in this figure, the miss traffic generated by the *real* compiler is higher than the dynamic adaptive scheme due to the marking of all array references as *wrt-inv* at compile-time. Nontheless, the *ideal* case indicates that the total network traffic can be potentially lower than that generated by the dynamic adaptive scheme.

As shown in Figure 3, a cache block size greater than a single word increases the total network traffic due to the false sharing effect. A comparison of the *real* compiler-assisted scheme with the invalidating scheme indicates that the total network traffic is comparable



(a) Only Scalar references

(b) Scalar and Array references

Figure 2: Total network traffic.

to that generated by the invalidating scheme while reducing the miss traffic and, therefore, the miss ratio. The lower miss traffic produced by the *real* compiler-assisted scheme is primarily due to the switching between updating and invalidating.

4 Conclusion

The compiler-assisted adaptive scheme can generate lower miss traffic and, thus, lower miss ratios, than the invalidating scheme while reducing the total network traffic to below that generated by the updating scheme, in all of the test programs. As a result, we conclude that the performance of a multiprocessor system can be improved by using the predictive ability of the compiler to select the best cache coherence protocol for each memory reference.

References

[1] A. Agarwal and A. Gupta. Memory-reference characteristics of multiprocessor applications under MACH. *Int. Sym. on Computer Architecture*, pages 215–225, 1988.

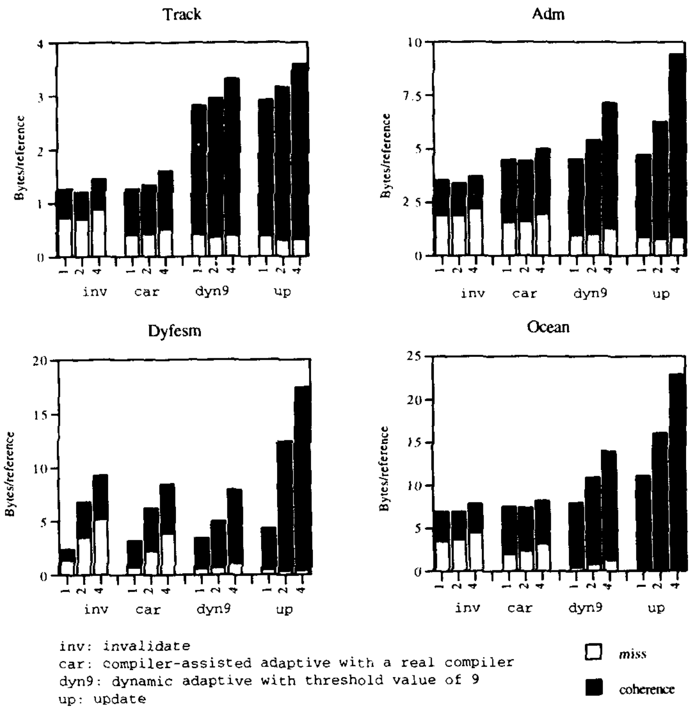


Figure 3: Total network traffic for array and scalar references with a 4-way set-associative 2Kword cache and cache block sizes of 1, 2, and 4 word.

[2] J. B. Andrews, C. J. Beckmann, and D. K. Poulsen. Notification and multicast networks for synchronization and coherence. *J. Parallel and Distributed Computing*, 15(4):332-350, August 1992.

[3] J. K. Archibald. A cache coherence approach for large multiprocessor system. In *Int. Conf. on Supercomputing*, pages 337-345, 1988.

[4] J. K. Archibald and J. Bear. An economical solution to the cache coherence problem. *Int. Sym. on Computer Architecture*, pages 355-362, 1984.

[5] J. K. Bennet and et al. Willow: A scalable shared memory multiprocessor. In *Supercomputing '92*, pages 336-345, November 1992.

[6] J. K. Bennett, J. B. Carter, and W. Z. Waenepoel. Adaptive software cache management for distributed shared memory architectures. *Int. Sym. on Computer Architecture*, pages 125-134, 1990.

[7] M. Berry and et al. The PERFECT Club benchmarks: Effective performance evaluation of supercomputers. Technical Report CSRD-827, U. of Illinois Urbana, May 1989.

[8] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache coherency schemes. *IEEE Trans. on Computers*, C-27(12):1112-1118, December 1978.

[9] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Int. Conf. on Architectural Support for Prog. Languages and Operating Systems*, pages 224-234, 1991.

[10] R. Cytron and J. Ferrante. What's in a name? or the value of renaming for parallelism detection and storage allocation. In *Int. Conf. on Parallel Processing*, pages 19-27, August 1987.

[11] S. J. Eggers and R. H. Katz. Evaluating the performance of four snooping cache coherency protocols. *Int. Sym. on Computer Architecture*, pages 1-15, 1989.

[12] K. Goshe and S. Simhadri. A cache coherence mechanism with limited combining capabilities for min-based multiprocessors. In *Int. Conf. on Parallel Processing*, volume I: Architecture, pages 296-300, 1991.

[13] A. Gupta and W. Weber. Analysis of cache invalidation patterns in multiprocessors. *Int. Sym. on Computer Architecture*, pages 243-455, 1989.

[14] A. Gupta, W. Weber, and T. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Int. Conf. on Parallel Processing*, volume I: Architecture, pages 312-321, 1990.

[15] A. R. Karline and et al. Competitive snoopy cacheing. *Annual Sym. on Foundations of Computer Science*, pages 244-254, October 1986.

[16] D. Lenoski and et al. The Stanford DASH multiprocessor. *Computer*, pages 63-79, March 1992.

[17] Z. Li. Array privatization for parallel execution of loops. In *Int. Conf. on Supercomputing*, pages 313-322, July 1992.

[18] F. Mounes-Toussi and D. J. Lilja. The potential of compile-time analysis to adapt the cache coherence enforcement strategy to the data sharing characteristics. *submitted to IEEE Trans. on Parallel and Distributed Systems*, 1993.

[19] N. Oba, A. Moriwaki, and S. Shimizu. Top-1: A snoopy-cache-based multiprocessor. In *Phoenix Conf. on Computer and Comm.*, pages 101-108, March 1990.

[20] C. D. Polychronopoulos and et al. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *Int. Conf. on Parallel Processing*, volume II: Software, St. Charles, IL, August 1989.

[21] A. W. Wilson and R. P. LaRow. Hiding shared memory reference latency on the Galactica Net distributed shared memory architecture. *J. Parallel and Distributed Computing*, 15(4):351-367, August 1992.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

The Impact of Cache Coherence Protocols on Systems Using Fine-Grain Data Synchronization

David B. Glasco, Bruce A. Delagi and Michael J. Flynn

Computer System Laboratory, Stanford University, Stanford, CA 94305

Abstract: In this paper, the performance of a fine-grain data synchronization scheme is examined for both invalidate-based and update-based cache coherent systems. The work first reviews coarse-gain and fine-grain synchronization schemes and discusses their advantages and disadvantages. Next, the actions required by each class of cache coherence protocols are examined for both synchronization schemes. This discussion demonstrates how invalidate-based cache coherence protocols are not well matched to fine-grain synchronization schemes while update-based protocol are.

To quantify these observations, five scientific applications are simulated. The results demonstrate that fine-grain synchronization always improves the performance of the applications compared to coarse-grain synchronization when update-based protocols are used, but the results vary for invalidate-based protocols. In the invalidate-based systems, the consumers of data may interfere with the producer. This interference results in an increase in invalidations and network traffic. These increases limit the possible gains in performance from a fine-grain synchronization scheme.

Keyword Codes: B.3.3; C.1.2
Keywords: Performance Analysis and Design Aids; Multiple Data Stream Architectures (Multiprocessors)

1 Introduction

In shared-memory multiprocessors, data is of'en shared between a producer and one or more consumers. To prevent the consumers from using stale or incorrect data, the consumers must not access the data until the producer notifies them that it is available. Typically, such systems use a coarse-grain synchronization scheme to synchronize this production and consumption of data.

In such schemes, simple flags can be useo to indicate that a given block of data has been produced. For example, the producer of a data block first writes the data to a shared buffer and then issues a fence instruction that stalls the processor until all writes have been performed. Finally, the producer sets the synchronization flag. The consumers, who have been waiting for the synchronization flag to be set, see the flag set and begin consuming the data. Coarse-grain schemes may also use other synchronization methods such as barriers. The coarse-grain synchronization schemes examined assume a release consistency memory model [2].

A coarse-grain synchror'zation scheme has two basic disadvantages. First, the scheme requires an expensive synchronization operation such as a flag or barrier to synchronize the production and consumption of data. Second, the consumers are forced to wait until the entire data block has been produced before they are able to begin consuming the data.

An alternative to the coarse-grain synchronization scheme discussed above is a fine-grain scheme. In such a scheme, the synchronization information for each data word is combined with the word. In this case, the producer creates the data and writes it to a shared buffer. The consumers wait for the desired word to become available and then consume it.

Fine-grain synchronization has several advantages. First, the consumers are able to consume data as soon as it is available. This allows the consumption time of the available data to overlap the production time of the subsequent words. Moreover, no expensive synchronization operation is required; this means that the producer never needs to wait for the writes to be performed.

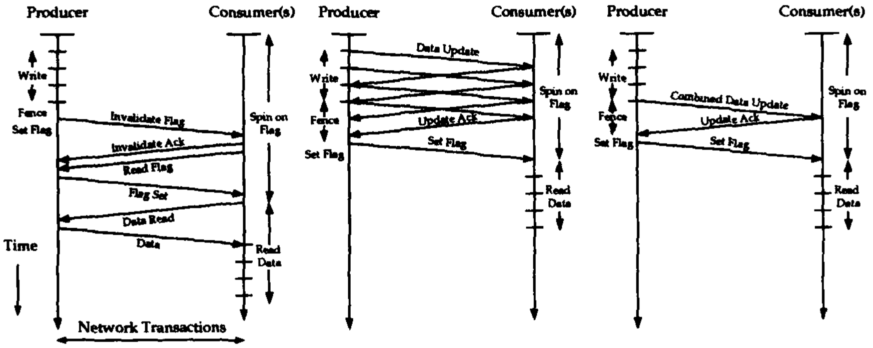
In this work, we are interested in studying the performance gains from a fine-grain synchronization scheme on scalable, cache coherent shared-memory multiprocessors. We will show that fine-grain synchronization may improve performance of applications running on systems with either invalidate-based or update-based cache coherence protocols, but that fine-grain synchronization is not a robust solution for invalidate-based systems while it is for update-based systems.

The paper is organized as follows. Section 2 describes a typical coarse-grain synchronization schen.e and demonstrates the resulting work required by each class of cache coherence protocols. Next, section 3 describes a fine-grain synchronization scheme and demonstrates how the scheme overcomes the disadvantages of the coarse-grain scheme. Section 4 describes the simulated architecture, the scientific applications and the cache coherence protocols examined in this work. Section 5 presents the simulation results. These results compare the performance of fine-grain synchronization to coarse-grain synchronization, and they also compare the relative performance of the cache coherence protocols when fine-grain synchronization is used. Finally, section 6 concludes the paper.

Our work is the first to examine the performance of fine-grain synchronization on both update-based and invalidate-based cache coherent multiprocessors. The work on the Alewife [6] system examines the implementation details of a fine-grain synchronization scheme, and their work gives an excellent description of the software and hardware requirements for such a scheme. In their work, they demonstrate that a fine-grain synchronization scheme may improve the performance of the given applications running on an invalidate-based cache coherent system. Our work does not contradict their findings, but rather expands them to demonstrate the instability of the combination of invalidate-based cache coherence protocols and fine-grain data synchronization.

2 Coarse-grain data synchronization

Figure 1a shows the actions required for invalidate-based protocols using a coarse-grain synchronization scheme. After producing the data, the producer writes it to a shared buffer and waits for the writes to be performed. The writes are considered performed when the producer's cache has obtained exclusive ownership of the line (all necessary invalidations have been performed) and the data has been written into the cache. The figure assumes that the producer has already obtained exclusive ownership of the data lines. After the writes have been performed, the producer sets the synchronization flag. If the consumers have already read the flag, then this write must invalidate the consumers' copies of the flag. The consumers, who are still waiting for the flag to be set, will immediately reread the flag after it is invalidated, but the producer cannot release the flag's line until all the invalidations have been performed. Once the consumers see the flag set, they can begin reading and consuming the data.



a) Invalidate-based protocols

b) Update-based protocols
Single word updates

c) Update-based protocols
Grouped updates

Figure 1: Coarse-grain synchronization

Figure 1a illustrates the cost of a coarse-grain synchronization scheme using a simple flag. The invalidation and reread of the flag by each consumer requires four network transactions and the transfer of a line of data. The work to transfer the synchronization information is more than the work to transfer one line of data. The size of the data block could be increased to reduce this synchronization overhead, but the larger block size also increases the waiting time of the consumers.

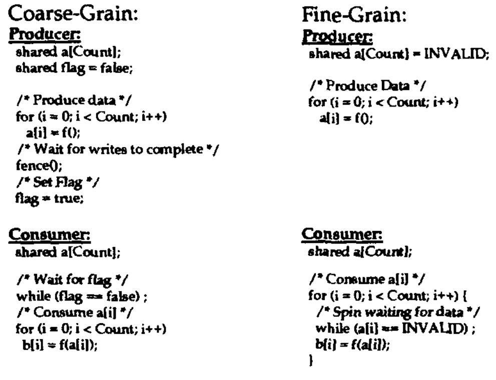
Figure 1b shows the actions required for update-based protocols. In this case, all of the consumers have prefetched the synchronization flag and data block before the data is written. When the producer writes the data, the consumers' caches are updated. The producer must wait for the writes to be performed (all updates acknowledged) before setting the synchronization flag. The producer's write of the flag results in the consumers' caches being updated. When the consumers see the flag updated, they can begin using the data, which is already in their caches. Figure 1c shows the resulting network transactions if a write-grouping scheme, as described in our earlier work [5, 4], is used. In this case, the data updates are grouped into larger, more efficient updates.

As in the invalidate-based case, the coarse-grain synchronization scheme has disadvantages. First, the cost of the coarse-grain synchronization operation is high. But the cost is not in transferring the synchronization information itself, it is in waiting for the updates to be performed (acknowledged). This synchronization scheme also forces the consumers to wait until the synchronization point is reached before accessing the data, but, unlike the invalidate-based case, the desired data is already in the consumers' caches as a result of the earlier updates. The coarse-grain synchronization scheme does not allow the system to take advantage of these fine-grain updates.

3 Fine-grain data synchronization

A fine-grain synchronization scheme would overcome many of the disadvantages of the coarse-grain scheme described in the last section. In a fine-grain synchronization scheme, the synchronization information is combined with the data. Such a scheme may be implemented in either hardware or software. In a hardware-based scheme, a full/empty bit is associated with each memory word [8]. Alternatively, a software-based scheme may be used in which an invalid code, such as NaN in a floating point application, is used to indicate an empty word. Currently, all the applications under study use a software-based scheme.

Figure 2 demonstrates how a producer and consumer interaction might be coded for both coarse-grain and fine-grain (using a software-based scheme) data synchronization. For the coarse-grain case, a simple flag, initialized to false, is used to synchronize the production and consumption of data. For the fine-grain case, the data is initialized to an invalid code. The consumer waits for each word to become valid and then consumes it. For iterative applications, the data must be set to the invalid code between iterations.



Coarse-Grain:

Producer:

```
shared a[Count];
shared flag = false;

/* Produce data */
for (i = 0; i < Count; i++)
  a[i] = f();
/* Wait for writes to complete */
fence();
/* Set Flag */
flag = true;
```

Consumer:

```
shared a[Count];

/* Wait for flag */
while (flag == false) ;
/* Consume a[i] */
for (i = 0; i < Count; i++)
 b[i] = f(a[i]);
```

Fine-Grain:

Producer:

```
shared a[Count] = INVALID;

/* Produce Data */
for (i = 0; i < Count; i++)
  a[i] = f();
```

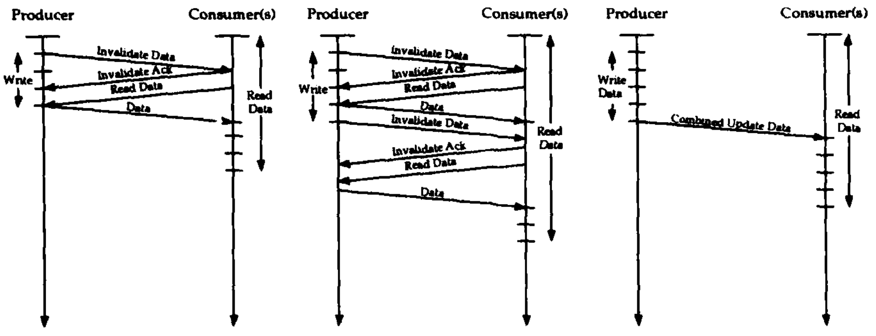
Consumer:

```
shared a[Count];

/* Consume a[i] */
for (i = 0; i < Count; i++) {
 /* Spin waiting for data */
 while (a[i] == INVALID) ;
 b[i] = f(a[i]);
}
```

Figure 2: Code examples for coarse-grain and fine-grain synchronization

By their very nature, invalidate-based protocols are not well matched to fine-grain synchronization schemes. The protocols do not allow consumers to maintain copies of a data line while a producer writes to the line. This results in a very unstable solution. Figure 3a shows the ideal timing diagram for invalidate-based protocols when a fine-grain synchronization scheme is used. First, all the consumers read the first word of the data block. When the producer writes this word, all the consumers' copies must be invalidated. But since the consumers are eagerly waiting for the data, they will immediately reread the data line once it is invalidated. The invalidate-based protocols studied allow the producer to continue writing into the cache line while invalidations are pending. This prevents the producer from observing any write delay as the consumers read and reread the line. In the ideal case, the invalidation latency is greater than the producer's write time for the line. When the consumers reread the line, they will find the line completely written. The producer will not invalidate the line again; this gives the consumers all the time they need to consume the line's data for this ideal case.



a) Invalidate-based protocols
Ideal case

b) Invalidate-based protocols
Consumer interference

c) Update-based protocols
Grouped updates

Figure 3: Fine-grain synchronization

However, figure 3b demonstrates the problem with invalidate-based protocols and fine-grain synchronization schemes. In this case, the consumers have reread the line after the initial invalidation but before the producer has completed writing the data. Now the producer is required to invalidate the consumers' copies of the line again, and the consumers are forced to reread the line. The producer is able to release the line again only after all the pending invalidations have been performed.

The relative timing of the writes and reads will have an enormous impact on the performance of the system. If the writes occur in bursts, as they often do, the producer will usually be able to produce many words of data between each reread by the consumers. But the consumers, who will reread the line immediately after it is invalidated, will only receive the data after all the consumers' copies of the line have been invalidated. The consumers will then be able to consume data only until the producer invalidates the line again, which may occur soon after the consumer receives the data if the write rate is high.

The frequency with which these invalidations and rereads occur depends on two characteristics of the application. First, the probability that the producer finishes writing the line before the consumers attempt to reread it depends on the number of words written to the line. This measure, known as the line utilization, will be used to classify the application space. The other characteristic is the number of consumers. The more consumers, the higher the probability that consumers will interfere with the producer's writing of the data.

The definition of update-based protocols offer a better match to fine-grain synchronization schemes. Unlike the invalidate-based protocols, update-based protocols allow consumers to maintain a copy of the data line while a producer writes to the line. Also, the producer's write of the data results in an update of all the consumers' caches: a proactive distribution of data rather that a reactive approach as in the invalidate-based protocols in which each consumer is responsible for refetching an invalidated line. For example, figure 3c shows the actions required for update-based protocols using fine-grain data synchronization and write grouping. First, all the consumers prefetch the desired data lines. As the producer writes the data, the writes are grouped and sent to the consumers, and the consumers can consume the data as soon as it arrives.

Update-based protocols can also take advantage of the fine-grain synchronization scheme in another way. Update-based protocols using a coarse-grain synchronization scheme require that updates be acknowledged before the synchronization flag is set, but in a fine-grain synchronization scheme, no update acknowledgment is needed because the producer is never required to wait for the updates to be performed. This has the largest impact on the distributed-directory update-based (DD-UP) protocol described in our earlier work [4].

The update-based protocols offer a much more robust solution. The data prefetch cannot degrade the performance of the fine-grain synchronization scheme as it can with the invalidate-based protocols [4]. The prefetch can be issued as early as desired by the consumers. Also, the relative timing of the producer's writes and consumers' reads can not affect performance as in the invalidate-based protocols. *The amount of work is fixed regardless of any variations in the timing of reads or writes.*

4 Simulation methodology

The Care/Simple simulation environment [1] was used to simulate a set of scientific applications running on a shared-memory multiprocessor. The simulated architecture consists of 64 nodes arranged in an 8 by 8 mesh. Each node consists of a processor/memory element (PME) connected to its four nearest neighbors through a set of network queues. A PME consists of a processor, cache, directory/memory and network interface. The processor is a 100 MHz superscalar processor that is assumed to be load/store limited, and the cache is a fully associative cache with infinite size. The cache has a single cycle access time and a line size of 16 words. Each memory consists of a single bank of 100 MHz synchronous DRAMs supporting page mode operation. The SDRAMs have 30 ns access time for a page access with a page miss penalty of an additional 60 ns. The directory consists of a 10 ns access time SRAM for all protocols. The network is order preserving with static, wormhole routing and multicast.

The applications studied here include a simple, iterative partial differential equation solver (PDE), a 3-D iterative partial differential equation solver using FFTs (3DFFT), and three different methods of factorizing a matrix into triangular matrices: a multifrontal solver (MF), sparse Cholesky factorization (SPCF), and LU decomposition.

Table 1 summarizes the important characteristics of the applications studied. The number of consumers for each data block gives a measure of general contention for each object and the maximum number of invalidates or updates that might be needed when the data is modified. The line utilization is the percentage of each memory line that is modified by the producer. For example, if the data is a dense vector, the producer is likely to modify all the words in the given line. This would result in a line utilization of 100%. If the data is a structure, the producer might only modify a few words, which would result in a low line utilization. In the invalidate-based protocols using fine-grain synchronization, these measures give an indication of the possible interference between the consumers and producer of a line of data. The larger the line utilization or number of consumers, the higher the probability of interference and extra invalidation and reread cycles. The table also indicates the type of the coarse-grain synchronization: a simple flag or barrier. The table specifies which applications are iterative and shows both the number of synchronization events in each application and the average number of words protected by each synchronization event.

The update-based protocols examined in this paper include the centralized-directory update-based protocol (CD-UP) and the distributed directory update-based protocol

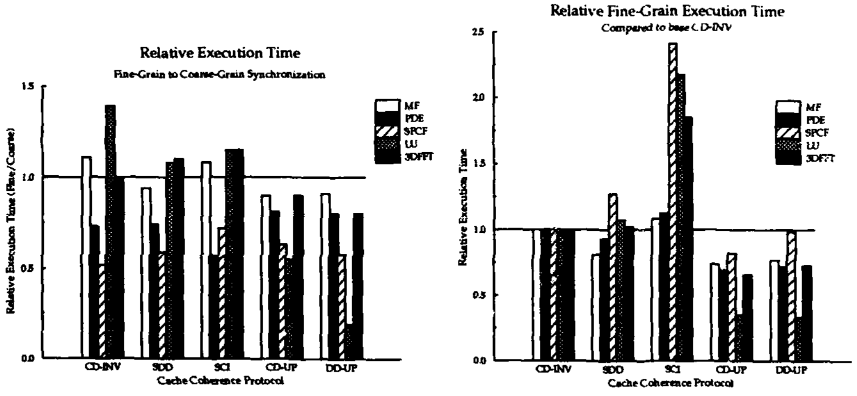
Applications	MF	PDE	SPCF	LU	3DFFT
Data Set (words)	1000x1000	32x32	1138x1138	64x64	8x8x16
Consumers	1	1	1.89	31.5	4
Line Utilization %	93.0	50.0	11.2	59.0	50.0
Sync Type	Flag	Flag	Flag	Flag	Barrier
Iterative	No	Yes	No	No	Yes
Sync Events	332	1120	2118	2016	678
Words/Sync Event	49.8	8.0	4.9	44.4	18.6

Table 1: Application characteristics

(DD-UP) described in our earlier work [4, 3]. The update-based protocols use the write-buffer grouping scheme also described in our earlier work [5]. The invalidate-based protocols examined include a centralized directory invalidate-based protocol (CD-INV), which is similar to DASH [7], a singly-linked distributed directory invalidate-based protocol (SDD) [9] and a doubly-linked distributed directory invalidate-based protocol (SCI), which is the IEEE standard protocol.

5 Results

The relative performance of the fine-grain synchronization scheme compared to the coarse-grain scheme and to a common base (CD-INV) is illustrated in figure 4. Table 2 gives the ratio of invalidations (updates) required and the relative change in total network traffic for the fine-grain synchronization case compared to the coarse-grain case for the invalidate-based (update-based) protocols.



a) Fine-grain execution time compared to coarse-grain

b) Execution time compared to common base (CD-INV)

Figure 4: Performance of fine-grain synchronization

For the invalidate-based protocols, the performance of the fine-grain synchronization scheme varies. For the applications with small blocks and few consumers (PDE and

	Invalidate-Based Protocols						Update-Based Protocols			
	Ratio of invalidations			*Ratio of network traffic*			Ratio of updates		Ratio of network traffic	
	CD-INV	SDD	SCI	CD-INV	SDD	SCI	CD-UP	DD-UP	CD-UP	DD-UP
PDE	0.93	0.93	0.93	0.67	0.55	0.51	1.85	1.91	0.87	1.09
SPCF	0.94	0.97	0.95	0.41	0.51	0.50	0.73	0.73	0.40	0.44
MF	10.7	7.08	4.07	0.81	0.81	0.80	0.98	0.84	0.84	0.84
3DFFT	1.00	1.08	1.14	0.98	1.07	0.99	1.38	1.38	0.84	1.13
LU	4.42	3.67	3.42	1.24	1.32	1.59	0.40	0.41	0.34	0.71

Table 2: Ratio of invalidations/updates and network traffic

SPCF), the fine-grain synchronization scheme improves the performance of the applications compared to the coarse-grain synchronization case. In these applications, the coarse-grain synchronization operation is costly, as each synchronization point protects only a small block of data: 8 words for the PDE application and 4.9 words for the SPCF application, as summarized in table 1. Therefore, the elimination of this coarse-grain synchronization operation outweighs any extra invalidations or network traffic generated by the fine-grain scheme. The fine-grain synchronization has actually reduced the total number of invalidations compared to the coarse-grain case for these two applications, as illustrated in table 2 for all the invalidate-based protocols. With the small block sizes of these applications, the producer was able to write the full block before the consumers could reread the line after the initial invalidation. The single invalidation per block acted as a synchronization or triggering event. The elimination of the explicit synchronization also significantly reduced the network traffic for these applications, as shown in table 2. This resulted in an improvement in execution times for the PDE and SPCF applications, as shown in figure 4a for the invalidate-based protocols.

As the line utilization increases, the consumer and producer interference also increases. For the MF application, the number of invalidations was small for the coarse-grain synchronization case. This indicates that the consumers were not always eagerly consuming the data. The fine-grain synchronization increased the number of invalidations significantly, but the overall traffic was reduced since the extra invalidation traffic was less than the traffic eliminated by the elimination of the explicit coarse-grain synchronization events. The actual execution time increased for the CD-INV and SCI protocols, but decreased for the SDD protocol. The difference in execution time between the invalidate-based protocols arises from the particular producer-consumer interaction that was interfered with. For the SDD protocol, the interference was off the critical timing path of the application and in the critical path for the other two invalidate-based protocols.

For the 3DFFT application, the iterative nature of the application required approximately twice the number of shared writes for the fine-grain synchronization case as compared to the coarse-grain case because the data must be set to an invalid code between iterations. As shown in table 2, these extra writes created at least as many invalidations as were eliminated by the fine-grain synchronization, and the resulting network traffic remained almost constant for the same reason. The small number of consumers increased the execution time of the distributed directory invalidate-based protocols (SDD and SCI) slightly more than the centralized-directory protocol (CD-INV). Overall, the fine-grain synchronization scheme did not improve the execution time for the 3DFFT application when invalidate-based protocols were used.

As the number of consumers and the line utilization increased, the consumer and producer interference also increased. For the LU application, fine-grain synchronization increased the number of invalidations and network traffic, as illustrated in table 2. With relatively inexpensive coarse-grain synchronization in this application, the increase in invalidations and network traffic outweighed any performance gains from the elimination of the coarse-grain synchronization operations. Again, fine-grain synchronization offered no improvement in execution time for systems with invalidate-based protocols.

For the update-based protocols, fine-grain data synchronization always improved the performance of the applications, as illustrated in figure 4a. The fine-grain synchronization decreased both the number of updates and the network traffic for non-iterative applications (SPCF, MF and LU), as shown in table 2. For the iterative applications (PDE and 3DFFT), the extra writes to clear the data between iterations increased the number of updates for both update-based protocols. The network traffic was reduced for the CD-UP protocol, but it increased slightly for the DD-UP protocol.

The fine-grain synchronization scheme had the largest impact on the DD-UP protocol when the number of consumers was greater than one (SPCF, 3DFFT and LU). In these applications, the coarse-grain synchronization scheme required the producer to wait for the updates to be propagated down the list of caches and then acknowledged. This limited the performance of the DD-UP protocol; the fine-grain synchronization scheme removed the need for these acknowledgements.

Figure 4b shows the relative execution time of the applications using fine-grain synchronization compared to the centralized directory invalidate-based protocol (CD-INV). As shown in the figure, the update-based protocols always perform better than any of the invalidate-based protocols when a fine-grain data synchronization scheme is used.

For the invalidate-based protocols using fine-grain data synchronization, the SDD protocol performed better for the applications with a single consumer. In these cases, the memory write backs of the CD-INV protocol were unnecessary since the data was not read from memory again because cache-to-cache transfers were used to transfer the data to the single consumer. But as the number of consumers increased, the invalidation latency of the distributed directory protocols resulted in longer execution times compared to the base CD-INV protocol.

For the update-based protocols, the performance of the two protocols was almost identical except for the SPCF application. The improvement in performance compared to the base CD-INV protocol was best for applications with high line utilization and a large number of consumers (LU). As the number of consumers decreased, the improvement from the update-based protocols also decreased compared to the CD-INV protocol. Compared to the SDD protocol, the improvement was less for applications with a single consumer, but it was more for applications with multiple consumers.

6 Conclusions

In summary, the fine-grain data synchronization scheme, when used with invalidate-based cache coherent systems, did not offer a robust solution. It only improved the performance for applications with a small number of consumers (less than 2) and a low line utilization. These application characteristics tended to avoid consumer interference. On the other hand, systems with update-based protocols could take advantage of the fine-grain synchronization scheme. The resulting execution times were always less than the coarse-grain synchronization case, and they were always less than the corresponding execution times for the invalidate-based systems using fine-grain synchronization.

The performance of fine-grain synchronization was unstable when invalidate-based cache coherence protocols were used because the definition of the invalidate-based protocols prevented the producer and consumers from actively sharing a memory line. The producer was required to obtain exclusive ownership of the line before writes could be performed. Consumers who might be consuming data from the line were forced to give up their copy of the line. The simulated results of the fine-grain scientific applications demonstrate the performance loss that can occur as the producer and consumers interfere with each other.

The definition of update-based protocols offered a much better match to the fine-grain data synchronization. The protocols allow multiple producers and consumers to maintain copies of a given memory line at the same time; the producer and consumers of data can not interfere with each other. Overall, our earlier work and this work have demonstrated the performance gains that can be achieved with update-based cache coherence protocols.

References

[1] Bruce A. Delagi, Nakul P. Saraiya, Sayuri Nishimura, and Gregory T. Byrd. Instrumented Architectural Simulation. In *Proceedings of the Third International Conference on Supercomputing*, pages 8–11, March 1988.

[2] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip. Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, 1990.

[3] David B. Glasco, Bruce A. Delagi, and Michael J. Flynn. Design and Validation of Update-Based Cache Coherence Protocols. Technical Report CSL-TR-94-613, Computer Systems Laboratory, Stanford University, March 1994.

[4] David B. Glasco, Bruce A. Delagi, and Michael J. Flynn. Update-Based Cache Coherence Protocols for Scalable Shared-Memory Multiprocessors. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 534–545, January 1994.

[5] David B. Glasco, Bruce A. Delagi, and Michael J. Flynn. Write Grouping for Update-Based Cache Coherence Protocols. Technical Report CSL-TR-94-612, Computer Systems Laboratory, Stanford University, March 1994.

[6] David Kranz, Beng-Hong Lim, David Yeung, and Anant Agarwal. Low-Cost Support for Fine-Grain Synchronization in Multiprocessors. In *International Symposium on Computer Architecture*, 1992.

[7] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

[8] Burton J. Smith. Architecture and Application of the HEP Multiprocessor Computer System. *Real Time Signal Processing IV*, 298:241–248, August 1981.

[9] Manu Thapar, Bruce A. Delagi, and Michael J. Flynn. Linked list cache coherence for scalable shared memory multiprocessors. In *7th International Parallel Processing Symposium*, April 1993.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Towards a Programming Environment for a Computer with Intelligent Memory

Abhaya Asthana, Mark Cravatts and Paul Krzyzanowski

AT&T Bell Laboratories
Murray Hill, New Jersey, 07922, USA

Abstract: We describe an experimental computing system based on an active memory architecture and its programming environment. The key idea is to directly embed within each memory unit some processing logic that can be programmed, at run time, to perform user defined operations on the data stored within the memory unit. Such an active memory architecture provides a natural and efficient framework for object oriented programs by directly supporting objects in memory and providing the underlying hardware base for a high performance storage server. Additionally, its unique memory-based approach to performing I/O enables application programs to have direct access to high speed network or disk data, with no involvement of the operating system kernel.

Keyword Codes:B.1.0;B.3.0;B.4.0;C.1.0;D.1.0;D.4.0
Keywords:Memory Architecture, Parallel Processing, Compilers, Object Oriented Programming

1. INTRODUCTION

For the past few years our group's research efforts have been focussed on memory participative computer architectures, in which the traditional ''passive'' role of the memory is superseded with memory as an active agent that cooperates with the main CPUs to complete a computation. To develop a deeper understanding the design and use of such systems, we have a built a prototype intelligent memory system at Bell Labs called SWIM (Structured Wafer-based Intelligent Memory.)

In this paper we briefly describe the architecture of the system and share our experiences in developing programming tools and applications for it. We present a snapshot of an on-going research project, reporting on what we have done so far, our current status, and plans for addressing outstanding issues/areas.

The demands of the target applications - primarily from telecommunications and databases - has considerably influenced our thought process. We believe that the existing *processor-centric* designs are not ideally suited for database and communications applications that by their nature tend to be memory intensive. For example, in communications processing, typically, data from a communication link gets deposited in memory through the system bus with the help of an I/O channel processor or a DMA unit. The processing of this data involves simple operations such as checksum computation, bit extraction, insertion, header parsing, link list manipulation, table look up, keyword searches and the like. Generally, no massively CPU intensive floating point vector operations are involved. Quite often (such as in routing or switching) the received data after some processing is put on an output queue for transmission back on the communication link. The situation in network query processing is similar. The data stream is divided into units (messages or packet) whose processing is repetitive, pipelined and localized. These properties are not well exploited in processor centric design of existing computer systems.

On the other hand, memory is a vastly underutilized resource in computer systems. Moreover, in its traditional form, it is not capable of scaling along with the CPU (the system bus just gets more congested). Large latencies in accessing data from the main memory to the CPU cause serious inefficiencies in many applications. Lack of fast context switching also limits the performance of real-time applications. In addition, locking operations for shared data structures with "passive" memories is another drain on the system bus.

2. SWIM ACTIVE MEMORY

We have developed an architectural solution based on active memories to address these issues [3]. The key idea is to take a small part of the processing logic of the CPU and move it directly inside every memory unit. Additionally, that processing logic can be programmed, at run time, to perform user defined operations on the data stored within a memory unit. By doing so, *the memory is no longer just a passive repository of data, but can actively participate in completing a computation along with the host CPU.*

Conceptually, SWIM appears as a high bandwidth, multiported memory system capable of storing, maintaining, and manipulating data structures within it, independent of the main processing units. The memory system is composed of up to thousands of small memory units, called *Active Storage Elements* (ASEs), embedded in a communication network. Figure 1 shows a conceptual picture of SWIM. Each ASE has *on-line* microprogrammable processing logic associated with it that allows it to perform select data manipulation operations locally. The processing logic is specially designed to efficiently per'orm operations such as pointer dereferencing, memory indirection, searching and bounds checking. This makes SWIM well suited to performing operations such as record searches, index table lookup, checksum computation and exception processing in active databases.

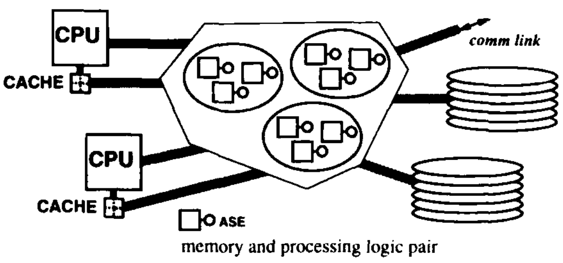


Figure 1. Conceptual Model

The memory system can be partitioned to support small and large objects of different types, some that fit within a single ASE, others that span several ASEs as shown in Figure 1. In the latter case, ASEs cooperate with each other to implement user defined distributed data structures and methods. ASEs can also connect directly via back-end ports to disks and communication lines.

Physically, the ASE represents a close coupling between the memory cells and specialized processing logic in the same local domain, thereby creating the potential for performance benefits. Figure 2 shows a block diagram of an ASE. The major components are the data memory itself, a 32 bit ALU with on-line microprogrammable control unit, and a two-ported switching and bus interface unit. The row and column bus interfaces allow ASEs to

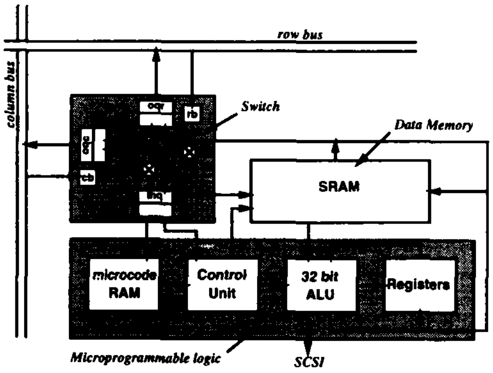


Figure 2. SWIM Active Storage Element

be connected in a two dimensional array. The switching logic supports the routing and buffering of messages between ASEs.

3. SWIM ARRAY

ASEs can be configured in various topologies to construct a parallel memory subsystem. Our experimental system has a 2-D array structure as shown in Figure 3 and consists of a 4x4 array of ASEs on a single VME card with SUN/SPARC as its host platform. Since the entire bus interfacing and switching logic is built within the ASE, no additional "glue" circuitry is required to construct a SWIM array. The ASE array interfaces to the host system bus through circuitry known as the CLAM (Control Logic for Access to Memory).

The Swim array appears as ordinary *memory* to the host processor but provides back-end interfaces to communication lines and to disk subsystems. Such a back-end connection into memory elements has several benefits. First, Messages from the communication lines can be received and processed entirely within this *active memory system* or be moved directly to the disk subsystem. Additionally, queries for data on the disks can be processed rapidly and replies forwarded to the network directly with virtually no involvement of the host processor. Second, since the ASEs handle all the low level data transfers and processing (such as filtering, format conversions, compression, decompression etc) much of the traffic is contained within the memory system. Third, the CPU(s) are relieved of much of the interrupt handling and context switching overhead associated with I/O transfers. Fourth, the architecture is scalable in that I/O bandwidth can grow with the size of the array. The memory size and processing capacity grows correspondingly with it. Last, multiple parallel I/O paths can be used to enhance either performance or fault tolerance or both.

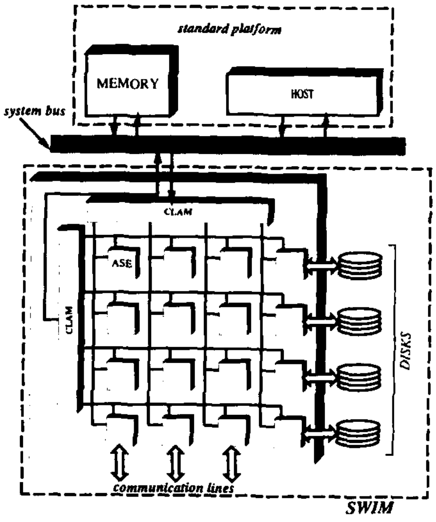


Figure 3. SWIM 4x4 Array

Other reasons to favor this model are technological. Memories themselves are getting denser but no more functional. A 64 Mbit chip is capable only of storing and retrieving a single word at a time. Adding 80,000-100,000 transistors of processing logic to that chip amounts to less that 0.1% of the chip's area, yet the functionality of the chip is now vastly increased. Secondly, because more computation is done on-chip and pad boundaries have to be crossed less often, a saving in power consumption results.

4. PROGRAMMING MODEL

An active memory provides a natural and efficient framework for object oriented programming by directly supporting objects in memory. This is significant because much of the investment for large networks is in software. If we consider an object-oriented programming paradigm, different memory processors can be programmed with the methods (member functions) to manage the objects for which they are responsible. Much of the computation can now be off-loaded onto the memory system. Memory functionality is increased to better balance the time spent in moving data with that involved in actually manipulating it. The host processor now has only the job of dispatching tasks to the memory processors (or object managers).

Performance gains are realized in several ways. First, the memory processor is tightly coupled with the memory. There is no slow system bus, and access is at cache speeds. Secondly, the instruction set can be optimized for "memory intensive" operations. Finally, concurrency is possible between memory processors, and between the host processor and a memory processor. Unless one processor needs the results or data of another, there is no reason why they cannot execute their programs asynchronously.

In an object-oriented framework, We can logically view a SWIM ASE as a microprogrammed implementation of a class. An object physically consists of a data structure and some member functions, shared by all objects of that class. The member functions are resident in the microcode of the ASE and the data resides in the ASE's data space as shown in Figure 4. An ASE provides the complete logical encapsulation of the object it manages. An object could be a data structure such as a priority queue or a graph, or an I/O object such as a disk or a communication link. External application level agents accessing the objects need not know about its internal implementation. Only the ASE concerned need have knowledge about the structure and semantics of the object, be it a queue, a communication link or a disk block server.

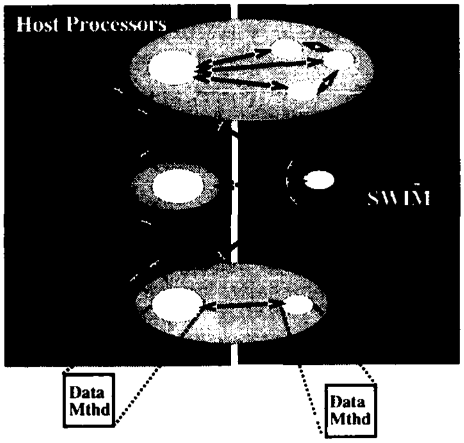


Figure 4. Cooperating Objects Model

Multi-ASE-Data Structures: Simple and small objects can be stored and manipulated entirely within an ASE, while larger and more complex objects are stored within several ASEs, and are cooperatively managed. Presently, the task of partitioning a large data structure and the assignment of ASEs to object data and code segments is up to the programmer. However, the architecture provides built-in support at the hardware level for low latency communication between ASEs (two cycles for sending a packet between ASEs), direct invocation (scheduling) of code sequences with the arrival of an ASE transaction, programmable traps, and a fast reply mechanism. Efficient fine grain parallelism can be achieved using these mechanisms.

5. BUILT-IN SUPPORT MECHANISMS FOR OBJECT MODEL

A unique feature of SWIM is its memory mapped architecture. The individual data memories *within each ASE are directly mapped into* the host CPU's address space and can be asynchronously read and written by it any time, as indicated by the thin arrows from application C to the ASE in Figure 5. This makes setting up of data structures, downloading object code, and communication between the host and ASE extremely easy. Since the entire state of the program running in SWIM is directly observable and controllable by the host, it also simplifies the debugging and monitoring of multi-ASE programs. We have developed a graphical parallel debugger for SWIM to ease the task of inspecting and debugging application programs. Packet transfers and reads and writes of an ASE's data address space become simple memory reads and writes.

Communication Modes: The architecture supports two modes of inter-ASE communication. The first is a three word packet form, for short asynchronous inter-ASE transactions. The notion of a message consisting of multiple packets is also supported by the hardware. The second form of communication is synchronous bulk data transfers between ASEs. Since ASEs typically communicate with each other using small messages, we designed a two level bus structure for the interconnect. Such a structure provides the smallest average latency of communication between ASEs for moderate size arrays, is simple to implement, and makes it easier to incorporate message broadcasting.

An ASE is configured with the member functions for a particular object class [6] by loading its on-chip microcode memory with the appropriate microcode to execute member functions associated with that class. In the SWIM system, this microcode can be *downloaded at run-time*. Conceptually, the data memory of the ASE is divided into multiple object buffers. Each object buffer is a chunk of memory large enough to hold all the data for an instance of the particular class. The memory system can be partitioned to support small and large objects of different types, some that fit within a single ASE, others that span several ASEs. In the latter case, ASEs cooperate with each other to implement user defined distributed data structures and methods.

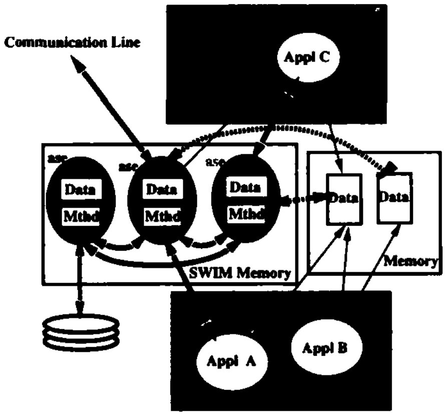


Figure 5. SWIM and Host Processor Interactions

A member function is invoked on a specific object by sending a message to the ASE managing it. Shown, for example, by the bold arrow from application C to the rightmost ASE. This message must identify the particular object of interest, the specific function to be executed, and the values for any parameters that may be required. Any response from the ASE is also in the form of a message. The SWIM system provides a very efficient mechanism (in the order of a few micro-instructions) to effect the transfer of short messages of this kind. The ASEs communicate with each other using this mechanism. Under the supervision of a host program, an ASE can independently read/write data from/to host's regular memory, process it on-the-fly, and move it to a specified I/O device. The dotted lines in Figure 5 show an example of such a transfer in which the middle ASE serves as a ''smart'' DMA agent to move data between host's regular memory and a communication line.

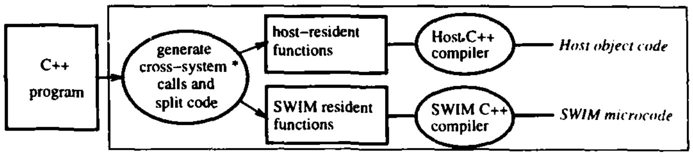
The processing between the receipt of a function invocation by an ASE, and the subsequent provision of the corresponding response back to the invoking entity is called a *transaction*. In the course of the execution of a transaction, multiple additional functions may be invoked. The entire computation has associated with it a system-generated transaction identifier, and the invoking entity can look for the associated results using this identifier. For interfacing with the host, the hardware provides multiple logical buffers at the output of SWIM, each associated with a different transaction identifier.

The function invocation mechanism is asynchronous in that there is no need for the invoking entity to wait while code for the invoked function is being executed. Thus, the overhead of member function invocation on an ASE is a few memory operations as far as a source ASE is concerned, and a few micro-instruction cycles as far as the target ASE is concerned. This overhead may cancel any benefits arising from executing code in SWIM rather than in the host for very trivial member functions. But, for any substantial member function, and especially if it is likely that a significant portion of the data is not cached, executing the member function on SWIM will be a win. However, note that individual memory read/writes by the host are synchronous — but these are usually quick enough so that the waiting time is not large.

Branches and programmable traps: Branches in an ASE are free; a conditional or unconditional branch consumes no additional cycles and may be incorporated into a microinstruction which performs other useful work. The *traps* in SWIM are programmable. In traditional processors, one operation that often consumes CPU cycles is checking for special conditions. Loops can often be coded tightly except for memory bounds checking. Counters are cheap to implement except that code has to check for a maximum or minimum value. When matching patterns, a tight loop is possible except that the pattern string may contain special characters or the data stream may contain special matches. All these conditions are handled in SWIM through an efficient trap mechanism. Traps may be set for memory bounds, maximum increment value, decrement reaching zero, and so on. When a trap condition occurs, program control is transferred to a user-specified address containing the trap handler. This mechanism, along with a long instruction word architecture, permits the coding of very tight loops (e.g. pattern matching can be performed in a single microinstruction).

6. C LANGUAGE SUPPORT FOR SWIM

While the SWIM ASEs may be programmed in microcode assembler, they are more conveniently programmed using the C programming language. The SWIM C compiler which generates microcode directly. At a higher level, classes in a C++ program can be targeted to reside on an ASE through pragma statements. The entire C++ program is then run through a compile system which splits code into host-resident and SWIM-resident sections. Method invocations are replaced with references to SWIM's memory. The SWIM resident portions are then compiled by the SWIM compiler and the host-resident portions are compiled by the standard C++ compiler on that host. If necessary, time-critical code segments can be specified using in-line assembly statements. Traditionally, microcoded architectures have been programmed through hand-microcoding. However, as parallel architectures will be increasingly used by programmers not intimately familiar with the details of the microarchitecture and software support and maintenance becomes more important, the ability to program the machine in a higher language becomes more desirable.



**cross-system calls are mechanisms by which the host communicates with SWIM (and SWIM communicates with the host).*

Figure 6. C++ Compilation on SWIM

The C programming language [5] was chosen as the input language for SWIM. It offers several advantages. For programming data-intensive operations, C allows the direct description of bit manipulations and low-level operations. It is a good language for program generation. C is already known and used by a large community of users. Because C is the dominant language on the host systems to which SWIM is currently interfaced, a C compiler on SWIM would be an ideal complement. Finally, programs written for an ASE may be tested on other C compilers on other machines.

A high level language may be supported on a microprogrammable machine in several ways. The first is to run an interpreter for the language in microcode. The second is to run an interpreter for an intermediate language in microcode. The compiler will translate the high level language into this intermediate language. The third, and last, is to compile the high level language directly into microcode. Banerji and Raymond [7] advocate the second method, arguing that microcode store is often too small and the translation may be easier if a good intermediate language is chosen. We opted for the third method instead, compiling directly into microcode to maximize flexibility and performance.

Most retargettable C compilers (e.g. *pcc*, UNIX's portable C compiler [1] and *gcc* from the Free Software Foundation) are inadequate for generating code for a SWIM ASE. Their model of the architecture, registers, and instruction space is too rigid to be retrofitted into a compiler for SWIM. In particular, they assume a von Neumann architecture with shared program and data memory, a single stack, and a single program counter. SWIM's microcode forces the compiler to deal with memory access through only a few memory address registers and a simulated stack. This is a great burden on automatic code generation. Common practices, such as allocating three memory address registers to serve as a frame pointer, argument pointer, and stack pointer are prohibitively expensive. The *lcc* compiler [4] has been chosen as a front end for a SWIM C compiler. It provides a convenient coupling between the front and back ends. It conforms to ANSI C and is fast and compact. The front end is responsible for parsing the language and produces a forest of directed acyclic graphs (dags) representing each compiled function. The back end selects code and annotates these dags and passes them back to the front end which then calls the back end to emit the code.

The goals of the compiler are to generate good, accurate, and compact code. The microstore on an ASE is not large, so the generated code is optimized and the microinstructions well-packed while ensuring that the behavior does not deviate from the programmer's intent. Moreover, facilities are provided to allow the knowledgeable programmer the ability to better control code generation. These controls include the generation of function prologues and epilogues, the mapping of variables to registers, and adding in-line assembly code. The pragma mechanism in C enables the support of these directives.

There are three steps in generating code for a SWIM ASE. The first is generating the dags in the front end and performing global optimization on the dags such as common subexpression elimination. The second takes place at register allocation time and involves allocating registers and annotating and/or restructuring dags to better fit SWIM's instruction set. The third stage takes place during code emission and consists of generating instructions, rewriting them, and packing a microword. In some cases instructions may be rewritten to better fill the microword. For instance, a register move may be rewritten to an equivalent alu operation if the alu fields are not in use, enabling another register move to take place in the same microinstruction cycle.

A set of functions and macros are available on the host end to facilitate interfacing with SWIM ASEs. Some of these facilities are functions to map SWIM's memory onto the processes memory space, claim a lock on SWIM, dereference ASE memory and register space to physical memory, and send and receive packets.

7. APPLICATION EXAMPLES

A number of applications have been written to explore the capabilities of SWIM and to obtain a measure of the potential performance gains. Results from a national phone database server, implemented on the existing SWIM system, indicate that it could handle in excess of 75 Million queries per day for retrieving a name/address given a phone number. We have also applied SWIM for prototyping a call screening application. The call screening agent receives copies of signaling messages and determines whether they should be processed by a service processor instead of being given normal treatment. In order to do this, it performs a database lookup on calling number and/or called number in real-time, and identifies the service processor where the features for the call are enabled. We showed that a query processing rate of 10,000 - 12,000 queries/second with nominal latency of 344 microseconds, is achievable in our lab prototype.

Gigabit IP Router

An IP packet router that can keep up with gigabit per second packet rates has been implemented on SWIM. Our approach is not to build a communications processor to support a special protocol set but, instead, to identify generic, commonly occurring communications processing functions and provide an implementation that efficiently supports these underlying functions in SWIM [2]. The motivation is similar to that in the use of DSP chips for signal processing applications in preference to regular general purpose microprocessors. Many of the communications processing functions are data-intensive, and most data-intensive processing is best done where the data is, in the memory system itself, independent of the main processing unit.

The three primary operations performed in a router are: 1) reception and transmission of the data frames from and to the link, 2) for an incoming packet deciding the outgoing link on which it should be transmitted, and, 3) switching the packet from the input link to the output link. Consistent with this functional division, the architecture shown in Figure 7 separates the data movement function from the actual function of routing based on the IP header. The latter function is performed by a SWIM/ASE based module, while the interface modules support various link protocols. The system can contain different types of link modules and multiple copies of a given type of link module.

The Interface Modules transmit and receive data from the links at the required bit rates. The data received from a link is saved in an input buffer. As a packet comes in, the IP header is stripped by the control circuitry, augmented with an identifying tag, and sent to an ASE for validation and routing. While the ASE is performing the routing function, the remainder of the packet is deposited in an input buffer in parallel. The ASE determines which outgoing link the packet should be transmitted on, and sends the updated header fields to the appropriate destination interface module a!ong with the tag information. The packet is then moved from the buffer in the source interface module to a buffer in the destination interface module and eventually transmitted on the outgoing link.

ASEs can each work on different headers in parallel. The circuitry in the interface modules peels the header off of each packet and assigns the header to ASEs in a round-robin fashion. Each ASE performs processing as discussed below. In some applications, order-maintenance is an issue. The output control circuitry also goes round-robin, guaranteeing that packets will then be sent out in the same order as they were received. (Better load-balancing may be achieved by having a more intelligent input interface which assigns each header tc the lightest loaded ASE. The output control circuitry would then have to select the next ASE to obtain a processed headers from by following the demultiplexing order followed at the input, so that order preservation of packets is ensured).

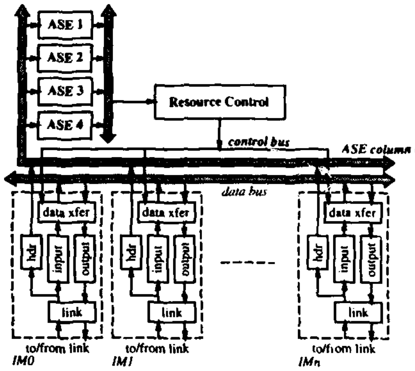


Figure 7. Organization of a SWIM based IP router

The performance results for processing an IP packet header varied depending on the type of packet. On a single 20 MHz ASE system, with 512 network addresses, a packet can be routed at a rate of 400,000 packets/sec . With host specific addresses and 2 fragments per packet, the speed falls to around 200,000 packets/sec.

Multiple ASEs can be used to obtain even higher throughput. Two header-processing ASEs provided a speed-up of 1.8-1.9 times that of a single ASE (the actual performance varied a little depending on the packet mix, and on other load on the system). With four ASEs, the speed-up obtained is 3.3-3.5. The less than linear speed-up shown is largely on account of the synchronization cost since we used a simple round-robin distribution mechanism, with essentially no buffering to even out the loads.

8. EXTENSIONS

We have discussed the suitability of an ASE managing a single class of objects. The scope of objects is great, in that they may vary greatly both in size and in quantity. Essentially, any size object can be supported by SWIM. If an object is so big that it cannot fit into the address space of a single ASE, one of two actions can be taken. Either parts of the object can be swapped onto a disk attached to the ASE or several ASEs can cooperatively manage different parts of the object. The choice is up to the programmer and depends on the size of the object, frequency of reference, and performance required.

A large number of objects can use up all available memory on an ASE. When this is a concern, a disk may be used as a store for swapping objects. This may be a local disk or another ASE with an attached disk devoted to object management. The advantage of a local disk is performance. The disadvantage is that code for managing a disk-based object store has to co-reside with the methods for the class that the ASE manages. On the other hand, a separate ASE acting as an object store can devote all of its resources to storing and caching objects. An ASE would send an object identifier to the object store and receive the object as a message. This latter approach is far more practical for all but extremely large objects where packet transmission times become significant.

Finally, in certain environments, persistent objects are needed. These are objects that live beyond the lifetime of a single process. An object store on SWIM can support such objects, using an ASEs memory for cache and the disk as a store. Note that while a process' address space disappears when the process dies, SWIM's ASEs and associated memory still remain active. Hence, even if an object is not placed in secondary storage, it still remains durable and persistence is achieved (without resorting to a file system).

9. CURRENT STATUS

A complete parallel hardware and software system constructed using an array of SWIM chips is operational. A photograph of the prototype SWIM board plugged into the VME bus on a Sun workstation is shown in Figure 8.

The chip, fabricated in 1.25 micron CMOS, has 80K transistors for logic and 4 Kbytes of downloadable microcode memory. We have not yet integrated the data memory into our experimental ASE, so the 512 Kbytes of data memory per ASE is currently external to the ASE chip.

Figure 8. SWIM Prototype Board

In order to use SWIM effectively, a suite of software tools to aid development and debugging has been written. This includes a microcode assembler, disassembler, compiler, graphical debugger, libraries, and over 50 user commands for manipulating various aspects of the SWIM system.

10. CONCLUSION

The results from the applications we have studied so far have been encouraging. We are exploring other applications of memory based I/O subsystems particularly in the area of wireless networks.

The performance gains realized by an active memory based system are due to three factors: 1) Data can manipulated in an ASE locally by the on-chip processor at a speed limited only by on-chip clock rate, and not by off-chip memory access times. 2) The ASE processing logic is designed to perform a generic class of data structure operations very well, resulting in several fold architectural performance advantage over a general purpose processor. 3) Both fine grain and medium grain parallelism in an application can be exploited using SWIM. Finally, the SWIM architecture scales with regard to processing power, memory size, memory bandwidth and I/O bandwidth.

Flexibility of design and on-line reconfigurability are additional advantages of our approach. This is significant because much of the investment for large networks is in provisioning, access and management of databases.

The SWIM C compiler has been used extensively. It produces good compact code within the realm of the ASE's architecture with which it is familiar. Through in-line assembly code, additional features of SWIM may be accessed. With pragmas, the programmer has more control over generated code. Interfacing between SWIM and host programs is made easy with include files produced by the compiler which define static variables for the host

SWIM's architecture, however, is far richer than that which can be easily compiled into a language such as C. In particular, the traps mechanism enables address or value bounds to be set and checked for automatically. When a condition is met, a trap takes place. These can be hand coded into SWIM C programs through a combination of in-line assembly code and register assignments. Ultimately, it would be desirable to have the compiler automatically detect conditions where using traps would prove useful. Similarly, SWIM's pattern matching hardware is unused by the compiler. Using it would involve putting a number of general purpose registers to special use and deducing segments of code that can be optimized by using this hardware.

REFERENCES

1. Aho. A., Sethi, R., Ullman, J., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Massachusetts, 1986.
2. Asthana, A., Delph, C., Jagadish, H.V., and Krzyzanowski, P., *Toward a Gigabit IP Router*. Journal of High Speed Networks 1, IOS Press, 1992, pp. 281-288.
3. Asthana, A. and Jagadish, H.V., *A High Bandwidth Intelligent Memory for Supercomputers*. Proceedings Third International Conference on Supercomputing, 1988.
4. Fraser, C.W. and Hanson, D. R., *A Code Generation Interface for ANSI C*, Princeton University Research Report CS-TR-270-90, October 1990.
5. Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, second edition, Prentice Hall, New Jersey, 1988.
6. Stoustrup, B. *The C++ Programming Language*. Addison Wesley, 1986.
7. Banerji, D. K. and Raymond, J., *Elements of Micro-Programming*, Prentice Hall, New Jersey, 1982.

PART IV

DISTRIBUTED MEMORY MACHINES

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Communication Analysis For Multicomputer Compilers

Inkyu Kim[a] and Michael Wolfe[b]

[a]Dept. of Computer Science, University of Oregon, Eugene, OR 97403, USA

[b]Oregon Graduate Institute, Beaverton, OR 97006, USA

Abstract: We use a general model to represent interprocessor collective communication in languages such as High Performance Fortran. Our model includes a communication pattern matrix and a shift vector. Most current research uses a simple distance vector [3, 7]; a distance vector corresponds to our shift vector when the communication pattern matrix is the identity matrix. We show how to find and operate on the communication pattern matrix from user-aligned array references. We also discuss the differences between physical and virtual communication. We have implemented some of our analysis methods in Tiny [11].

Keyword Codes: C.1.2; D.1.3; D.3.0
Keywords: Multiple Data Stream Architectures (Multiprocessors); Concurrent Programming; Programming Languages, General

1 Introduction

Multicomputers are an important parallel architecture because they are scalable and relatively inexpensive to build. However, multicomputer MIMD or SPMD programs are hard to develop and extremely difficult to debug. Several higher level languages have been proposed to help multicomputer programmers; SISAL, Crystal, Kali, Superb and Fortran D are some examples. Most, if not all, use an SPMD programming model that utilizes data parallelism, as described in [2, 6]. In these higher level languages, interprocessor communications are generated by the compiler [13].

In multicomputers such as the Intel iPSC/860 and nCUBE, the overhead time associated with sending a message is relatively high. Reducing the total communication overhead is an important optimization. One approach is to minimizing the number of communications by using "vectorization," that is, transmitting whole blocks of data rather than single elements. Current systems such as Fortran D [3] and Superb [7] use distance vector information, similar to dependence distances [1], to classify communication patterns. These patterns are used to vectorize communications. We believe that distance vectors are too restrictive for communication analysis.

In this paper, we propose a systematic approach to analyze interprocessor communication for multicomputers, assuming data decomposition is described by the user. We use a programming model similar to that of High Performance Fortran (HPF). We first apply

```
forall( $i_1$ = $LB_1$:$UB_1$, $i_2$ = $LB_2$:$UB_2$, ..., $i_n$ = $LB_n$:$UB_n$)
    X($i_{\pi_1}$,$i_{\pi_2}$,...,$i_{\pi_s}$) = $\beta$( Y($G_1$($i_1$,$i_2$,...,$i_n$),$G_2$($i$),...,$G_s$($i$)))
```

Figure 1: Language Model

usage analysis to find a *communication matrix* and a *shift vector* for every right hand side array reference in an array assignment. Communication patterns are classified for each array reference according to the pattern matrix. The analysis is used to find communication in the virtual space. The model proposed for this analysis has some severe restrictions, which we relax in the final section.

2 The Program Model and Definitions

The program model we are using is similar to the one used for data dependence analysis. We focus on parallel assignments, such as Fortran 90 array assignments or `forall` statements, as shown in Figure 1. For this paper, our examples have three restrictions:

1. The dimensionality of each array is equal to the number of parallel indices;
2. The subscript expressions in the left hand side array are a simple permutation of the parallel indices. This is easily satisfied by appropriate restructuring, as shown by Li and Pingali [4]; and
3. The right hand side subscript expressions $G(i)$ are linear combinations of the parallel indices, which is also typical of data dependence analysis methods.

We discuss these restrictions further in the last section.

2.1 Data, Iteration and Processor Space

In a multicomputer application, the programmer and/or compiler must consider three different indexed spaces, i.e., the data space, the iteration space, virtual process space and the actual processor space. We define these as follows:

- Data Space: Each array defines a data space, where each point corresponds to an element of the array.
- Iteration Space: The iteration space of a d-nested loop or `forall` is a d-dimensional Cartesian space, where each point corresponds to an iteration of the loop or `forall` body.
- Virtual Space: A virtual space is an abstract indexed space that roughly corresponds to `templates` in HPF. Data alignment is used to describe how points of a data space are aligned to points of a virtual space, as described in Li and Chen's work [5, 12] or in High Performance Fortran. When two points from different data spaces are aligned to the same point in a virtual space, they will always be allocated to the same physical processor, regardless of the distribution used.

Two points from the same data space will always be aligned to different points in virtual space. In this paper we assume a single virtual space, to which all data spaces are aligned. We define the *active area* of a virtual space as follows: a parallel assignment updates a subset of the data space of its left hand side; the points in virtual space to which that subset is aligned is called the *active area.*

- Physical Space: A physical space is an indexed space such that each point corresponds to a physical processor. Each processor has its own ID number. The topology of the interconnection network is not considered here, thus our analysis is architecture independent. The virtual space is distributed over the physical space, which implies distribution of the data spaces; when two points in virtual space are assigned to the same point in physical space, they are allocated to the same physical processor. Again we assume a single physical space, which may have fewer dimensions than the virtual space.

Data alignment and distribution define how the data spaces are distributed over the physical space. The goal is to allow users to write programs in a data distribution independent manner. The distribution can then be tuned without changing the program. When the distribution is known, the compiler can generate appropriate communications for each array reference in a parallel assignment.

2.2 Data Decomposition

Here we study only block decompositions of dimensions of the virtual space onto the physical space. Cyclic data decompositions [3] have not yet been considered. Possible *data decompositions* for a two dimensional virtual space are:

- Decomposed by row: `X(block,:)`.
- Decomposed by column: `X(:,block)`.
- Decomposed by subblock: `X(block,block)`

Decompositions for higher dimensional spaces can be described similarly. We have also studied diagonal decompositions, where `X(i,j)` and `X(k,l)` are allocated to the same physical processor if `i+j=k+l`, but these are not reported here.

3 Usage Analysis

For each right hand side use of an array that fits our model, we find a *communication pattern matrix M* and a *shift vector s* to represent the usage pattern. The matrix M and the shift vector s are simply obtained by representing the right hand side array subscript expressions as a function of the left hand side subscripts. Analyzing the matrix and shift vector tells us a great deal about the necessary communication. Our representation has advantages over other work, which concentrates on communication distance vectors, as we will demonstrate. In the following, upper case italics refers to matrices, and lower case italic names are column vectors.

```
forall( i = 1:n, j = 2:n )
    a(i,j) = a(i,j-1)
```

$$M = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, S = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

Figure 2: Identity communication pattern matrix.

```
forall( i = 1:n, j = 1:n )
    a(i,j) = a(i,i)
```

$$M = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}, S = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Figure 3: Singular (multicast) communication pattern matrix.

3.1 Mappings

The iteration space is indexed by the vector of loop indices, i. When we restrict the left hand side reference subscript expressions to be a permutation of the index vector, the LHS reference can be expressed as X(Pi), for some permutation matrix P. Each right hand side array reference can be represented by expressing the subscript functions in matrix notation, as Y($Gi + g$). We call G the *data access matrix* and g the *access offset vector*, extending Li and Pingali's work [4]. This gives us a mapping from each point in the iteration space to the corresponding points in the data space.

We also assume that the arrays are all aligned to a single virtual space by a simple affine mapping. Initially we study only *axis alignment* and *offset alignment* (i.e., no scaling). The mapping from a point d in a data space to the corresponding point v in virtual space is given by an axis alignment matrix A and an alignment offset vector a. Thus, $v = Ad + a$; for now we restrict ourselves to the case where A is nonsingular and unimodular (integer entries, determinant = ± 1).

In the virtual space, communication must occur from the RHS array to the LHS. The LHS array reference is at virtual point $w = A_X(Pi) + a_X$, and the RHS array reference is at virtual point $v = A_Y(Gi + g) + a_Y$. Solving for i in terms of w gives us

$$i = P^{-1} A_X^{-1}(w - a_X)$$

Inserting this into the source equation, we get commmunication to point w from point

$$v = A_Y(G(P^{-1} A_X^{-1}(w - a_X)) + g) + a_Y$$

Algebraic manipulation can give us communication to point w from point $v = M_{XY} w + s_{XY}$, where

$$\begin{aligned} M_{XY} &= A_Y G P^{-1} A_X^{-1} \\ s_{XY} &= A_Y(G(-P^{-1} A_X^{-1} a_X) + g) + a_Y \end{aligned}$$

Note that even in the simple case of communication with the same array or arrays that are identically aligned, the alignment matrices do not drop out of the expression.

3.2 Examples

To demonstrate the basic idea, we give some examples. In Figure 2, the communication pattern matrix is the identity matrix, since the LHS and RHS arrays are the same (and

```
template t(:,:)
align (a(i,j),t(i,j))
align (b(j,i),t(i,j))
forall( i = 1:n, j = 1:n )
     a(i,j) = b(i,j)
```

$$M = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, S = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Figure 4: Transpose communication pattern matrix, through alignment.

thus must be aligned), and the indices appear in the same order in the subscripts. Figure 3 shows a singular communication pattern matrix; this is a situation where each point in virtual space must send a value to many receivers. Because of the restrictions of our model, this is called a *linear* commmunication, since the virtual point of the source can be computed as a linear combination of the virtual indices of the destination. Here, each source has n destinations. In general, whenever the communication pattern matrix is singular, we have a linear communication. Figure 4 shows the matrix computed after taking into account the alignment of two arrays onto a common virtual space.

4 Virtual Communication

The virtual communication is communication between source and destination points in the virtual space. The communication pattern matrix and shift vector can be used to determine the source point in virtual space from the destination. For any point w in the active area, w will receive data from virtual point $v = Mw + s$.

4.1 Basic Idea

Here we describe a simple process to find the destination point or points from each possible virtual source. This uses a great deal of the linear algebra that is also used in analysis of subscript expressions for data dependence analysis, as in Banerjee's monograph [9]. Given the matrix and vector M and s as in the previous section, we wish to decide for each virtual point (1) whether that point is a source for this communication, and (2) what are the virtual destination points. Moreover, most of this work should be done at compile time, so that at run time the decisions can be made as efficiently as possible. We do this by finding a unimodular matrix U and a lower triangular integer matrix L such that $MU = L$. We can then solve $Lt = v - s$, where v is the index of the potential source point. Since L is lower triangular, a simple back substitution will suffice. This solves for t in terms of v; in case of linear communication, one or more of the t indices will be left unspecified (a free variable). If there is no integer solution, then this virtual point is not a source. If there is a solution, then $Lt = v - s$ can be rewritten as $MUt = v - s$ or $Mw = v - s$ where $w = Ut$. Thus the matrix product Ut gives the destinations in terms of t, where some or all of the t entries are replaced by their equivalents in terms of v. The proofs of this are in [9]. During the first reduction process (finding U and L), redundant rows of M will be eliminated, which will add constraints to the indices of the source points. For instance, if we find that row 2 of M is twice row 1, then we eliminate row 2 of M, and add the constraint that $v_2 = 2v_1$. The second element of v is then completely determined, and does not participate in the rest of the matrix solution.

```
forall( i = 1:n, j = 1:n )
     a(i,j) = a(i+j-1,2*i+2*j-3)
```

$$M = \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix}, S = \begin{pmatrix} -1 \\ -3 \end{pmatrix}$$

Figure 5: Linear communication pattern matrix.

In the rest of this section, we show how to find communication pattern using M.

4.2 Nonsingular Communication

In the example from Figure 4, we have

$$M = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, S = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

We can use any method to find appropriate L and U matrices; efficient methods are described in [9, 8]. The simplest solution here is

$$L = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, U = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

The reader can trivially confirm that $MU = L$. At any virtual point $v = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$ we can solve the matrix equation $Lt = v - s \ with \ t_1 = v_1, t_2 = v_2$. The destination point w is computed as

$$Ut = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} v_2 \\ v_1 \end{pmatrix}$$

This is an example of a nonsingular communication matrix with no redundant rows or columns and no free entries of t. We found other nonsingular communication patterns such as rotation, but thsese are not reported here.

4.3 Singular/Linear Communication

Consider the example in Figure 5. The methods used for the Power Test for data dependence [8] to find the L and U matrices will also determine that the second row of M is redundant. In the matrix equation $Mw = v - s$, the second row is eliminated by adding the constraint $v_2 - s_2 = 2(v_1 - s_1)$, or in this example, $v_2 = 2v_1 - 1$. We then find L and U for the reduced system as

$$L = \begin{pmatrix} 1 & 0 \end{pmatrix}, U = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$$

Again, the reader can confirm that $MU = L$. We solve $Lt = \tilde{v} - \tilde{s}$, where $\tilde{v}$ and $\tilde{s}$ are v and s reduced by eliminating the elements corresponds to the redundant rows of M; in this case L has one row and $\tilde{v}$ and $\tilde{s}$ have one element. The matrix equation solves to $t_1 = v_1 + 1$; since t_2 is not constrained by the solution, it is a free variable. From the matrix formula $Ut = w$, we find the destinations as

$$w = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} = \begin{pmatrix} t_1 - t_2 \\ t_2 \end{pmatrix}$$

Thus, a value of $t = (t_1, t_2)$ describes a communication with source at $(t_1 - 1, 2t_1 - 3)$ and a destination at $(t_1 - t_2, t_2)$. Given a virtual point at (v_1, v_2), it is a source point if $v_2 = 2v_1 - 1$; if so, it has destinations at all virtual points $(v_1 - t_2 + 1, t_2)$ for all values of t_2 such that the destinations lie within the active area. We can solve for the range of t_2 (or all free variables, in general) for which this virtual point has destinations by applying the boundary region constraints of the active area to the destination point solution. If the active area is $(1 : 10, 1 : 10)$, then virtual point $(3, 5)$ is a source, since $5 = 2 \times 3 - 1$, and its destinations are those points $(4 - t_2, t_2)$ that lie in the active area. From the first index we get the inequalities $1 \leq 4 - t_2 \leq 10$ or $-6 \leq t_2 \leq 3$; from the second index we get the inequalities $1 \leq t_2 \leq 10$. Combining these, we have the limits $1 \leq t_2 \leq 3$, so the destinations are $(3, 1)$, $(2, 2)$, $(1, 3)$.

5 Physical Communication

In the multicomputer system, many virtual points are clustered together on a single physical node. In this paper, we only consider simple block decompositions.

When a dimension of the virtual space is distributed, communication across that virtual dimension will cause physical communication. This can be easily tested by looking at the formulas for source and destination virtual points. In the example in Figure 3, we have a communication source at virtual point (v_1, v_2) when $v_2 = v_1$, and its destinations are those points (w_1, w_2) with $w_1 = v_1$, and w_2 is free. In these formulas, the first index of source and destination is the same, while the second differs. If only the first virtual dimension is distributed, this reference pattern will cause no communication in the physical space.

In the example in Figure 5, the communication source is virtual point $(v_1, 2v_1 - 1)$ with destinations at $(v_1 - t_2 + 1, t_2)$ for all valid values of t_2. Communication will occur no matter which dimension is distributed, since both indices of source and destination differ for some values of t_2. When there is no free variable in the formula for w_j, for some j, and only that dimension is distributed, then there will be only one physical destination for each virtual source point.

Three kinds of optimization can affect physical communication. When two or more virtual destination points for the same virtual source point are assigned to the same physical processor, the message needs to be sent only once; we call this *message collapsing.* The situation is recognized when there is a free variable (one of the t variables) in the formula for destinations in the distributed index, and its coefficient is less than the distribution block size. In the example from Figure 5, if the first dimension is distributed with a block size of 10, then each virtual source has up to ten virtual destinations in the same physical processor (since the coefficient of t_2 in the destination first index is one).

When two or more virtual source points are allocated to the one physical processor and have virtual destinations on the same physical destination processor, the messages can be collected and sent as a vector. Previous research has called this *message vectorization* [10], and focused on constant distance communication, when this is easy to recognize (the communication pattern matrix is the identity matrix).

Finally, and most importantly, when the source and destination virtual points are allocated to the physical processor, no message should be sent. We are working on the criteria that characterize these last two conditions.

6 Summary

We have described a method to classify references in data parallel languages that take into account global indices and individually aligned arrays. The method allows the compiler to generate code to compute the destinations for each data source. The classification is more general than previous research, which focuses only on fixed distance communication. It is related to Li and Chen's work [12], though that work attempts to automatically generate an alignment from a declarative algorithmic specification. We have implemented some of the compiler analysis in our research prototype, Tiny. We are now working on experimenting with some of the linear communication patterns on an Intel iPSC/860 multicomputer.

In our work here we have several severe restrictions on the programming model. One of the restrictions is that the left hand side array subscript expressions be a simple permutation of the parallel index variables. This is not too severe; by the rules of **forall** assignments (as proposed in Fortran 8x and used in High Performance Fortran) only one value can be assigned to any array element. Thus if the subscripts can be represented by the matrix formula $Fi + f$, the matrix F must be of rank n, where n is the number of parallel indices. If F is square, it must be invertible; the compiler can then invert the matrix and modify the index limits and right hand side references to satisfy this condition. For instance, the assignment

```
    forall( i = 1:N, j = 1:M )
        A(i,i+j-1) = B(i,j)
```

can be rewritten internally by the compiler as

```
    forall( i = 1:N, j = i:i+M-1 )
        A(i,j) = B(i,j-i+1)
```

More general restructuring is shown by Li and Pingali [4]. We use a slightly more general data access function than that paper, and generalize the programming model by allowing different alignment functions and composing the alignment and access functions to access the virtual space.

The restriction that the arrays have the same dimensionality as the number of parallel indices can also be relaxed. Arrays with greater dimensionality must have redundant rows in the access matrix. These are eliminated before (or during) processing, and added back after solving the reduced system. Arrays with less dimensionality cause other problems. On the left hand side, deficient arrays often correspond to manually linearized multidimensional arrays. On the right hand side, deficient arrays are less of a problem; when a vector is *replicated* across a virtual 2-dimensional space, the compiler has the freedom to choose which replicated copy of the vector to use as a source. Non-replicated deficient arrays will be allocated to a fixed subset of the virtual space, causing communication in some virtual dimension. These issues have not been fully investigated.

The final restriction to subscript expressions that are linear combinations of the parallel indices is typical of such analysis work. Nonlinear subscripts, such as indexed subscripts, must be handled dynamically or by some other means. We intend to focus on the important case where compile time analysis can be effective.

Future work will look at analyzing parallel programs to see what types of communication pattern matrices actually appear. We will also look at whether this analysis can be used to drive a program restructuring process, when the parallel assignment appears in sequential loops. In this situation, depending on the distribution, loop interchanging or other restructuring transformations may be able to optimize the communication pattern and frequency. We also plan to use this formulation to evaluate the cost of the communication by inspecting the regularity of the communication pattern and estimating the communication *density*, where the density is the maximum amount of data transferred from any source or to any destination.

Acknowledgements

The authors thank Akiyoshi Wakatani for his discussions and corrections to our manuscript.

References

[1] David A. Padua and Michael Wolfe, "Advanced Compiler Optimizations for Supercomputers", *Communications ACM*, Vol.29, No.12, pp. 1184-1201, 1986.

[2] Allan H. Karp "Programming for Parallelism", *IEEE Computers*, May , pp. 43-57, 1987.

[3] Seema Hiranandani and Ken Kennedy and Chau-Wen Tseng "Compiling Fortran D for distributed-Memory Machines", *Communications ACM*, Vol.35, No.8, pp.66-80, 1992.

[4] Wei Li and Keshav Pingali "Access Normalization: Loop Restructuring for NUMA Compilers", *Proc. Fifth International Conf. on Architectural Support for Program ming Languages and Operating Systems*,Oct., pp. 285-295, 1992 *ACM Press*

[5] Jingke Li "Compiling Crystal for Distributed-Memory Machines", Dept. Computer Science, Yale Univ. Ph.D Dissertation, Oct, 1991

[6] W.D. Hillis and G.L. Steele, Jr. "Data Parallel Algorithms", *Communications ACM*, Vol. 29, No. 12, pp. 1170-1183, 1986.

[7] H.P. Zima and H.J. Bast and H.M. Gerndt "SUPERB: A tool for semi-automatic MIMD/SIMD parallelization", *Parallel Computing*, Vol.6, pp. 3-18, 1988.

[8] Michael Wolfe and Chau-Wen Tseng "The Power Test for Data Dependence", *IEEE Trans. Parallel and Distributed Systems*, Vol.3, No.5, pp.591-601, 1992.

[9] Utpal Banerjee "Dependence Analysis for Supercomputing", *Kluwer Academic Publishers*, 1988.

[10] Anne Rogers and Keshav Pingali "Process Decomposition Through Locality of Reference", *Proc. ACM SIGPLAN '89 Conf. on Programming Language Design and Implementation* , pp.69-80, 1989.

[11] Michael Wolfe "The Tiny Loop Restructuring Research Tool", *Proc. 1991 International Conf. on Parallel Processing*,pp.46-53, 1991

[12] Jingke Li and Marina Chen "Compiling Communication-Efficient Programs for Massively Parallel Machines", *IEEE Trans. Parallel and Distributed Systems*, Vol.2, No.3, pp.361-376, 1991.

[13] David Callahan and Ken Kennedy "Compiling Programs for Distributed-Memory Multiprocessors", *The Journal of Supercomputing*, Oct. 1988, pp.151-169.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Automatic Data Layout Using 0–1 Integer Programming[1]

Robert Bixby[a], Ken Kennedy[b], and Ulrich Kremer[b]

[a] Department of Computational and Applied Mathematics, Rice University, U.S.A.

[b] Department of Computer Science, Rice University, U.S.A.

Abstract: The goal of languages like Fortran D or High Performance Fortran (HPF) is to provide a simple yet efficient machine-independent parallel programming model. By shifting much of the burden of machine-dependent optimization to the compiler, the programmer is able to write data-parallel programs that can be compiled and executed with good performance on many different architectures. However, the choice of a good data layout is still left to the programmer. Even the most sophisticated compiler may not be able to compensate for a poorly chosen data layout since many compiler decisions are driven by the data layout specified in the program.

The choice of a good data layout depends on many factors, including the target machine architecture, the compilation system, the problem size, and the number of processors available. The option of remapping arrays at specific points in the program makes the choice even harder. Current programming tools provide little or no support for this difficult selection process.

This paper discusses automatic data layout techniques in the context of a programming environment and an advanced compilation system that allows dynamic data remapping. Our proposed framework for automatic data layout builds and examines search spaces of candidate data layouts. A candidate layout is an efficient layout for some part of the program. Choosing a single layout for each program part among its candidate data layouts such that their overall cost is minimal has been shown to be NP-complete. Instead of resorting to heuristics, this paper investigates methods to determine the optimal selection. The data layout selection problem is formulated as a 0–1 integer programming problem, which is then fed to a state-of-the-art, general purpose integer programming solver. Our experiments show that even though we use a general purpose integer programming tool, there exists a formulation that can be solved very efficiently. Comparisons with similar 0–1 problems and their special purpose solvers indicate that our results can be improved on significantly as well if a special purpose solver is used.

[1] **Corresponding author**: Ulrich Kremer, Department of Computer Science, Rice University, 6100 S. Main, Houston, Texas 77005; e-mail:**kremer@cs.rice.edu**. This research was supported by the Center for Research on Parallel Computation (CRPC), a Science and Technology Center funded by NSF through Cooperative Agreement Number CCR-9120008. This work was also sponsored by ARPA under contract #DABT63-92-C-0038, and the IBM corporation. The content of this paper does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement should be inferred.

Keyword Codes: C.1.2; D.2.6; D.3.4
Keywords: Multiple Data Stream Architecture (Multiprocessors); Programming Environment; Processors - compilers, optimization

1 Introduction

The advent of languages like High Performance Fortran (HPF) [20], in which the programmer specifies parallelism implicitly by specifying the layout of an application's data across the processor array, has focused renewed attention on the problem of choosing a good data layout for parallel execution. Many experts believe that the choice of data layout is one of the two most important steps, in addition to choice of a suitable algorithm, toward a successful parallel implementation.

However, many programmers are not sure what data layout to choose. Many complex considerations must be taken into account if the program is to perform at high efficiency. For example, it is almost essential to consider the program as a whole rather than a series of independent subroutines. Of particular importance and complexity is the problem of determining when dynamic data redistribution will enhance overall performance.

Fortunately, the designers of languages like HPF and Fortran D [14], by requiring that data layout specifications be provided by the programmer, have opened the door for powerful new tools which can use intensive computation to determine a first approximation to a good data layout automatically. Because these tools are not embedded in the compiler and will be run only a few times during the implementation phase of a project, they can use techniques that would be considered too computationally intensive for inclusion in compilers, even on today's powerful supercomputers. Furthermore, by providing a high-level target language for these tools, HPF and similar languages have dramatically simplified the implementation of data layout tools.

In this paper, we will describe a data layout tool that uses a number of techniques from linear and integer programming. Integer programming is required because the automatic data layout problem that we solve is NP-complete. Evidence will be presented that our approach will be efficient enough for use in a programming assistance tool.

Our ability to solve integer programming problems has been remarkably improved over the last five to ten years, particularly pure 0–1 integer problems such as those being generated here. The basic technique for solving integer problems is to apply intelligent branch-and-bound using linear programming at the nodes. Important improvements have come in three areas. First, linear programming codes are on average approximately two orders of magnitude faster than they were five years ago, particularly for larger problems [6]. Combined with the improvements in computing speed over that same period these codes represent an approximate four orders of magnitude improvement in our ability to solve linear programming problems.

The second major development is in so-called cutting-plane technology. Motivated by work of Dantzig, Johnson and Fulkerson in the 50's [12], Padberg, Groetschel and others have shown how cutting-plane techniques could be used to strengthen the linear programming relaxations of many pure 0–1 integer programming problems [25]. The strengthening is effected by studying the facets of the underlying polytope generated by the convex hull of 0–1 solutions. Knowledge of these facets leads to subroutines for recognizing inequalities violated by the current fractional solution. These violated inequalities can then be added to the linear programming formulation in lieu of branching.

The third major area of improvement has come in the application of parallel processing to handle the branching when cutting planes do not succeed in sufficiently strengthening the linear programming formulation. Parallelism is particularly appropriate for current cutting-plane methods because cuts are computed not only at the root node but at all nodes in the branching tree. The extra computation at the nodes has the effect of making the computations sufficiently coarse grained that communication costs need not be significant. The most striking example of an integer programming success story exploiting all of the above advances is the recent work of Applegate, Bixby, Cook and Chvatal in which a 4461 city traveling salesman problem was solved to exact optimality using a complex branch-and-cut code running on a network of up to 60 loosely connected workstations [3].

2 Framework for Automatic Data Layout

The choice of a good data layout for a program depends on the compilation system, the problem size, the number of processors used, and the performance characteristics of the target machine architecture. Our proposed framework for automatic data layout will determine a good data layout for a given compilation system. Although this framework is not designed to be used inside the compilation system, it knows about the transformations and optimizations performed by the compilation system. The data layout is optimized for a target distributed-memory machine, a specific problem size, and the number of available processors, which implies that these entities have to be known at tool invocation time.

This section gives an overview of the proposed framework for automatic data layout for regular problems. Typically, regular problems represent data objects as dense arrays as opposed to a sparse representation. Regular problems allow the compilation system to determine the communication requirements and to perform a variety of program optimizations at compile time. The framework assumes that different data layouts can be specified for different program sections. Our proposed strategy for automatic data layout in the presence of dynamic data remapping consists of several steps, each discussed briefly in the remainder of this section. A detailed program example can be found in Section 3.

2.1 Overview

The initial step partitions the program into code segments, called program *phases*. Data remapping is allowed only between phases. A *phase* is the outermost loop in a loop nest such that the loop defines an induction variable that occurs in a subscript expression of an array reference in the loop body. This operational definition does not allow the overlapping or nesting of phases. Other strategies for identifying program phases are a topic of current research. Note that loop transformations such as loop fusion or loop distribution can change the partitioning of a program into phases. The discussion of transformations in the context of phase recognition is beyond the scope of this paper.

The phase structure of the program is represented in the *phase control flow graph*, an augmented control flow graph [1] where each phase is represented by a single node. The graph is annotated with branch probabilities and loop control information.

In the second step, the *program alignment space* is determined. The program alignment space corresponds to a single HPF template or a single Fortran D decomposition. All alignments and distributions are specified based on this unique program alignment space.

In the next two steps, data layout search spaces are constructed for each phase. A data layout for a single phase is specified by the alignment and distribution of all arrays referenced in the phase. First, alignment analysis builds a search space of reasonable alignment schemes for each phase. If arrays have fewer dimensions than the program alignment space, alignment analysis may generate different embeddings for the arrays.

Then, distribution analysis uses the alignment search spaces to build candidate data layout search spaces of reasonable alignments and distributions for each phase. A preliminary discussion of possible pruning heuristics and the sizes of their resulting search spaces can be found in [16].

After the generation of the search spaces, a single candidate data layout is selected for each phase, resulting in a data layout for the entire program. This step solves the so-called *inter-phase* data layout problem. The inter-phase data layout problem will be described in more detail in the next section. The selected candidate layouts have minimal overall cost. The overall cost is determined by the costs of each selected candidate layout, and the required remapping costs between selected candidate layouts. Note that the optimal data layout for a program may consist of candidate data layouts that are each suboptimal for their phases. The selection process is based on static performance estimates of the candidate data layouts and of data remappings between layouts. A static performance estimator suitable for automatic data layout has been discussed elsewhere [4, 14]. The performance estimator will mimic the compilation process for single phases. It will use the training set approach to determine communication and computation costs of the target machine architecture.

In the final step, data layout specifications are generated. The program alignment space is represented by a single **decomposition** or **template** statement. A data flow problem over the phase control flow graph is solved to avoid redundant specifications of alignments and/or distributions for the selected candidate data layouts. The final data layout specifications are translated into corresponding **align** and **distribute** statements.

2.2 Inter-phase Data Layout Problem

The inter-phase data layout problem is modeled as an optimization problem over the *data layout graph*. The data layout graph has one node for each candidate data layout. Edges represent possible remappings between candidate data layouts. Nodes and edges have weights representing the overall cost of each layout and remapping, respectively, in terms of execution time. The costs reflect the frequencies or probabilities of phase execution. A data flow problem over the phase control flow graph can be solved to determine these probabilities. The proposed data flow problem is similar to a reaching definitions problem [1] that additionally determines the probability that a definition reaches a given point in the program.

The initial formulation of the inter-phase data layout problem as an optimization problem over the data layout graph does not model the possible overlap of communication and computation between phases. However, the static performance estimator for each single phase will take the effect of compiler generated wavefronts inside a phase into account [27]. In addition, the initial formulation requires that only a single copy of an array can exist at any time during program execution, unless the array is replicated due to multiple ownership. This restriction can be relaxed by adding additional remapping edges. The placement of these remapping edges is a topic of current research.

The inter-phase data layout problem is proven to be NP-complete [21]. The proof is based on a reduction from the 3-CNF satisfiability problem (3-SAT) [11]. However, in the special case where each candidate layout specifies a mapping for every array in the program, the inter-phase data layout problem can be solved in polynomial time in the size of the data layout graph. The polynomial time algorithm uses dynamic programming to solve multiple single-source, shortest path problems over the data layout graph [22].

In this paper, we focus on methods to compute the optimal solution of the inter-phase data layout problem with only a single copy of each array at any time during the execution of the program. An instance of the inter-phase data layout problem is translated into a 0–1 integer programming problem suitable to be solved by *CPLEX*[2], a linear integer programming tool partly developed by Robert Bixby at Rice University [5]. We give experimental results for our 0–1 integer programming formulations for an 800 line benchmark code developed by Thomas Eidson at ICASE.

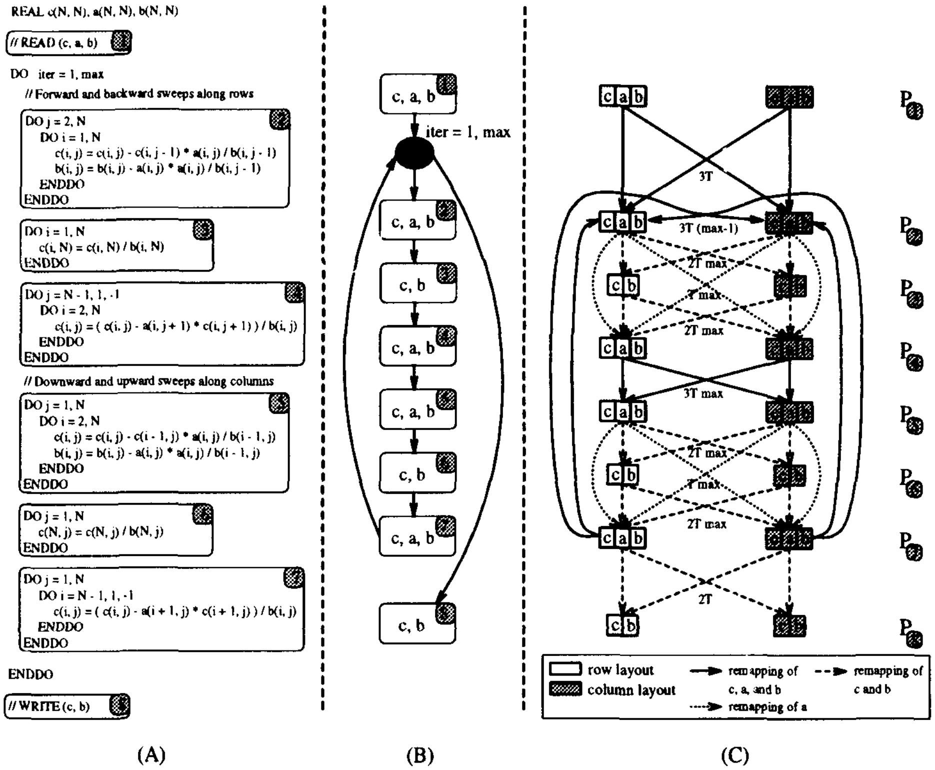
3 Example Program

The following example illustrates the framework for automatic data layout. Figure 1A shows an Alternating Direction Implicit (ADI) integration kernel. ADI integration is a technique frequently used to solve partial differential equations (PDEs). The operational phase definition partitions the code into eight phases (see Section 2.1). The first phase and the last phase are loop nests that perform input and output operations, respectively. The resulting phase control flow graph is shown in Figure 1B. The program has a two-dimensional alignment space of size **N** in each dimension. To simplify the example, we assume that alignment analysis builds alignment search spaces that map the three arrays **a**, **b**, and **c** canonically onto the program alignment space. To simplify the example even further, we assume that distribution analysis generates only two candidate data layouts for each phase, namely a row layout and a column layout. The resulting data layout graph is shown in Figure 1C. The edges represent possible remappings of arrays between candidate data layouts. To model the effects of the iterative loop, the data layout graph contains edges between the layouts of the seventh phase and the second phase. The node and edge weights are the estimated costs for phase execution and remapping, respectively, in terms of execution time, multiplied by their predicted execution frequencies. The cost for transposing a single array is denoted by **T**. **max** is the number of iterations in the iterative loop. A static performance estimator together with the computed phase execution frequencies will be used to determine the node and edge weights [4, 14].

To solve the inter-phase data layout problem, a single candidate data layout must be chosen for each phase such that the overall cost of the selected layouts is minimal. The overall cost of a set of selected layouts is the sum of the weights of their representing nodes and the weights of all edges between these nodes. The solution requires that the value of **max** is known.

The execution of the ADI integration kernel consists of a repeated sequence of forward and backward sweeps along rows, followed by downward and upward sweeps along columns. For the sweeps along the rows, a row layout has the best performance. The same holds for a column layout for the column sweeps. Transposing the arrays between the row and column sweeps, i.e. between phases 4 and 5, and between phases 7 and 2, eliminates communication within phases. In contrast, choosing the same data layout for both, row and column sweeps will avoid communication between phases but will make communication necessary inside some of the phases. The right choice will depend on the speed of the communication hardware and software of the target distributed-memory machine, and the ability of the compiler to exploit pipelined parallelism efficiently. In addition, the performance characteristics of the underlying I/O system has to be considered for the candidate layouts of the first and the last phase. Figure 2 shows two different data layout specifications that may be generated by the automatic tool. The left hand side shows a static, column-wise data layout. The right hand side depicts a dynamic data layout where transpose operations will be performed between the row and column sweeps.

[2] *CPLEX* is a trademark of CPLEX Optimization, Inc.



```
REAL c(N, N), a(N, N), b(N, N)

// READ (c, a, b)

DO  iter = 1, max
  // Forward and backward sweeps along rows

  DO j = 2, N
    DO i = 1, N
      c(i, j) = c(i, j) - c(i, j - 1) * a(i, j) / b(i, j - 1)
      b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i, j - 1)
    ENDDO
  ENDDO

  DO i = 1, N
    c(i, N) = c(i, N) / b(i, N)
  ENDDO

  DO j = N - 1, 1, -1
    DO i = 2, N
      c(i, j) = ( c(i, j) - a(i, j + 1) * c(i, j + 1) ) / b(i, j)
    ENDDO
  ENDDO

  // Downward and upward sweeps along columns

  DO j = 1, N
    DO i = 2, N
      c(i, j) = c(i, j) - c(i - 1, j) * a(i, j) / b(i - 1, j)
      b(i, j) = b(i, j) - a(i, j) * a(i, j) / b(i - 1, j)
    ENDDO
  ENDDO

  DO j = 1, N
    c(N, j) = c(N, j) / b(N, j)
  ENDDO

  DO j = 1, N
    DO i = N - 1, 1, -1
      c(i, j) = ( c(i, j) - a(i + 1, j) * c(i + 1, j) ) / b(i, j)
    ENDDO
  ENDDO

ENDDO

// WRITE (c, b)
```

Figure 1: ADI integration kernel with three array variables. A: Source code with phase partitioning. There are eight phases. The loops representing the first and the last phase are not shown. Each phase references either two or three arrays. B: The phase control flow graph for the phase partitioning. C: The data layout graph for the candidate data layout search spaces of the phases. To simplify the example, we assume that there are only two candidate data layouts in each search space. Weights represent static performance estimates of overall execution times. Node weights are not shown. Unlabeled edges have zero weight. T is the cost of performing a single array transpose, and max is the number of iterations of the outermost loop of the ADI integration kernel.

```
REAL c(N, N), a(n, N), b(N, N)

// Static column-wise layout
!HPF$ TEMPLATE X(N, N)
!HPF$ ALIGN c(i, j), a(i, j), b(i, j) WITH X(i, j)
!HPF$ DISTRIBUTE X(:, BLOCK)


      . . .
DO iter = 1, max
   // Forward and backward sweeps along rows

         . . .

   // Downward and upward sweeps along columns

         . . .

ENDDO
      . . .
```

```
REAL c(N, N), a(n, N), b(N, N)

// Dynamic row and column-wise layout
!HPF$ TEMPLATE X(N, N)
!HPF$ DYNAMIC X
!HPF$ ALIGN c(i, j), a(i, j), b(i, j) WITH X(i, j)
!HPF$ DISTRIBUTE X(:, BLOCK)
      . . .
DO iter = 1, max
   // Forward and backward sweeps along rows
   !HPF$ REDISTRIBUTE X(BLOCK, :)
         . . .
   // Downward and upward sweeps along columns
   !HPF$ REDISTRIBUTE X(:, BLOCK)
         . . .
ENDDO
      . . .
```

Figure 2: Example output of the proposed framework for automatic data layout for the ADI integration kernel. The left hand side shows a static column-wise data layout, and the right hand side shows a dynamic layout that performs transposes between the sweeps along rows and columns.

4 Related Work

The problem of finding an efficient data layout for a distributed-memory multiprocessor has been addressed by many researchers [2, 8, 9, 10, 13, 15, 17, 18, 19, 23, 24, 26, 28]. The presented solutions differ significantly in the assumptions that are made about the input language, the possible set of data layouts, the compilation system, and the target distributed-memory machine. Even though many researchers have recognized the need for dynamic remapping and are planning to develop solutions, our work is one of the first to provide a framework for automatic data layout that considers dynamic remapping.

Knobe, Lukas, and Dally [19], and Chatterjee, Gilbert, Schreiber, and Teng [9, 10] address the problem of dynamic alignment in a framework particularly suitable for SIMD machines. Anderson and Lam discuss techniques for automatic data layout for distributed and shared address space machines [2]. Their approach considers dynamic remapping. Lee and Tsai propose a dynamic programming algorithm to determine a data layout for a sequence of loop nests, allowing remapping between the loop nests [23].

In contrast to most of the previously published work, our framework is designed to work in the context of a programming assistance tool, not inside a compiler. As a consequence, the framework can use techniques that may be too expensive to be included in a compiler.

5 Inter-phase Data Layout as a 0–1 Problem

This section discusses the details of translating an instance of an inter-phase data layout problem and its data layout graph into an instance of a 0–1 integer programming problem with linear constraints, or *0–1 problem* for short. For the purpose of this discussion, we assume that the input program has n phases, $P_1, \dots P_n$. The corresponding data layout

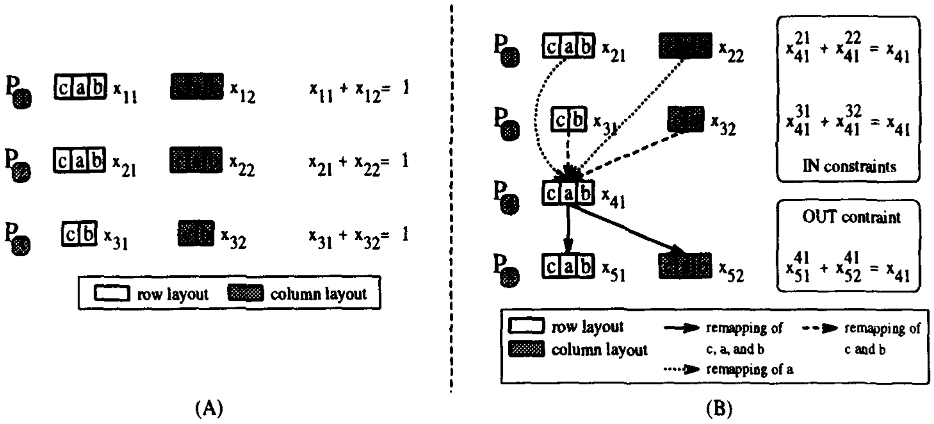


Figure 3: A: Layout constraints for the first three phases. B: Remapping constraints for the first candidate layout in the fourth phase. Switch x_{41} represents this layout.

graph has m_i nodes for each phase i, $1 \leq i \leq n$. Remember that each node represents a particular candidate data layout that specifies the alignment and distribution of all variables referenced in the phase.

An *instance of a 0–1 problem* consists of a set of variables X, a set of linear constraints over the variables in X, and a linear objective function with domain X. A solution to an instance of the 0–1 problem is a function $s_{01} : X \rightarrow \{0, 1\}$ that minimizes the objective function while respecting the constraints.

The translation introduces a variable for each node and edge in the data layout graph. These variables can be thought of as switches that are *on* if and only if the represented nodes or edges are part of a solution of the inter-phase data layout problem. The set X is the union of two sets of variables, $X = X_{layout} \cup X_{remap}$. X_{layout} contains a single switch for each node in the data layout graph, and X_{remap} has one switch for each edge. The switch $x_{ik} \in X_{layout}$ represents the k-th node of the i-th phase. The switch $x_{ik}^{jl} \in X_{remap}$ represents the remapping edge between the l-th node of phase j and the k-th node of phase i.

Similar to the variable set X, the set of constraints is partitioned into two classes. Constraints that ensure the selection of only a single node for each phase are called *layout constraints*. For each phase i, $1 \leq i \leq n$, there is a constraint of the form $\sum_{k=1}^{m_i} x_{ik} = 1$. In other words, exactly one switch has to be *on* for each phase and all other switches for the phase have to be *off*. The layout constraints for the first three phases of our example data layout graph (Figure 1C) are shown in Figure 3A.

Remapping constraints guarantee that all remapping edges between selected nodes are considered, i.e. are also selected. For each node in the data layout graph, there are two types of remapping constraints, namely IN-constraints and OUT-constraints.

For the node represented by x_{ik} and all incoming edges with nodes in the same phase j as their sources, IN-constraints of the form $\sum_{l=1}^{m_j} x_{ik}^{jl} = x_{ik}$ are generated. In other words, if switch x_{ik} is *on* then exactly one switch representing an incoming edge from phase j must be *on*. If x_{ik} is *off* then all incoming edge switches have to be *off*.

Similarly, for the node represented by x_{ik} and all outgoing edges with nodes in the same phase j' as their sinks, OUT-constraints of the form $\sum_{l'=1}^{m_{j'}} x_{j'l'}^{ik} = x_{ik}$ are generated. The remapping constraints for the node representing the first layout in the fourth phase of the ADI integration example, x_{41}, are listed in Figure 3B.

A solution s_{01} of an instance of the 0–1 problem *minimizes* the following objective function under the above constraints:

$$\sum_{x_{ik} \in X_{layout}} x_{ik}\ cost_{layout}\ (x_{ik}) \ + \ \sum_{x_{ik}^{jl} \in X_{remap}} x_{ik}^{jl}\ cost_{remap}\ (x_{ik}^{jl})\ ,$$

where $cost_{layout}$ and $cost_{remap}$ represent the node and edge weights of the data layout graph, respectively.

Note that we also investigated other valid formulations of the data-layout problem as a 0–1 problem. However, these other formulations turned out to be inferior to the presented formulation in terms of the time *CPLEX* needed to compute the optimal solution. A more detailed discussion of the different formulations can be found elsewhere [7].

6 Experiments

All of our experiments are based on ERLEBACHER, a 800 line benchmark program written by Thomas Eidson at the Institute for Computer Applications in Science and Engineering (ICASE). The program performs 3-dimensional tridiagonal solves using Alternating Direction Implicit (ADI) integration. The code contains computational wavefronts across all three dimensions. Array kill analysis was performed by hand and arrays were renamed and replicated appropriately. The resulting program contains 40 phases and 25 arrays. There are arrays with one, two, and three dimensions.

ERLEBACHER has a three-dimensional alignment space. For our experiments, we assume that alignment analysis and distribution analysis generate seven candidate layouts for each phase, one layout for each possible combination of distributed dimensions. However, if a phase contains only one-dimensional arrays, its candidate search space has only four layouts since some layouts are the projection of two distribution schemes. The corresponding data layout graphs with different weights were generated by hand. Weights were chosen to model different communication costs and the presence or absence of compiler optimizations. For instance, a compiler may be able to generate a coarse-grain pipelined loop if the data layout induces cross-processor dependences [27]. Whether the compiler performs such an optimization or not is represented by different weights given to the nodes.

We wrote a tool that generates the 0–1 problem formulation (see Section 5) for a given, weighted data layout graph. The remapping costs for individual arrays are handed to the tool as a cost table. The tool automatically generates the edge weights of the corresponding data layout graph based on this cost table. The generated 0–1 problems have 253 layout switches, 2133 remapping switches, and 715 layout and remapping constraints.

For the experiment, twelve 0–1 problems were automatically generated. Each of the 0–1 problems was solved by *CPLEX*, a linear integer programming tool. *CPLEX* includes an implementation of a general-purpose branch-and-bound code for mixed integer programming. Being general purpose, this code does not exploit the structural properties of our particular 0–1 problems. The experiments show that our 0–1 formulation can be solved by the general-purpose *CPLEX* in less than 4 seconds on average on a SPARC-10 workstation. For one 0–1 problem instance, *CPLEX* determined the optimal solution in 2.6 seconds. *CPLEX* did not take longer than 4.8 seconds on any of the twelve 0–1 problem instances.

0–1 integer programming is NP-complete. Therefore, it is unrealistic to expect a solution for all instances in minimal computation time. However, recent experience with other NP-hard problems formulated as 0–1 integer programs — principally the TSP — indicate that a careful study of structure of the particular integer program can lead to very effective practical procedures. Recent work on the TSP again provides a good example of what one can hope for. Using the well-known TSPLIB test set of problems, we have been able to solve to exact optimality all instances with fewer than 2000 cities, with the notable exception of one 225 city instance for which our cutting plane methods simply do not seem to be effective. However, it is interesting to note that for this one instance, it is symmetry that makes that problem difficult for our algorithms. That very symmetry implies that various heuristic procedures easily find the optimal solution (provably optimal by an independent analysis of problem structure). For the problem at hand, namely the inter-phase data layout problem, a similar approach is proposed in which inexact heuristic procedures would be applied if integer programming fails to find a solution within acceptable time limits. We remark in this context that we would expect our branch-and-cut algorithm to be computing both upper and lower bounds as it proceeds. If the computation is terminated prior to optimality, these bounds would provide estimates of the solution quality.

CPLEX is also designed to be applied as a callable library of linear-programming routines that can be conveniently built into a branch-and-cut code, such as a special purpose solver for the inter-phase data layout problem.

7 Future Work

So far, the discussion of the our proposed framework has ignored procedure calls. All steps of the framework as discussed in Section 2.1 will have to be performed interprocedurally. For the inter-phase data layout problem, we plan to use the following approach in the absence of recursion. The data layout graphs of subroutines can be propagated bottom-up along the edges of the call graph, resulting in a single data layout graph for the entire program. If the compilation system performs procedure cloning, a distinct copy of a procedure's data layout graph is propagated along each edge in the call graph. This strategy has been used to hand generate the data layout graph of the ERLEBACHER code used for the experiments discussed in Section 6. In the absence of cloning, each procedure is represented by a single copy of its data layout graph in the data layout graph for the entire program.

We are currently investigating the support of data replication in our framework. There are two forms of data replication. Data replication that consists of multiple copies of an array with distinct owners for each copy will be handled during the construction of the candidate search spaces, i.e. search spaces may contain candidate layouts that specify multiple owners of an array. Data replication that refers to multiple copies of an array with only a single owner can be modeled by additional remapping edges in the data layout graph. The IN and OUT-constraints of our presented 0–1 problem formulation will have to be slightly modified to accommodate the new edges. Additional constraints can be used to enforce an upper bound on the number of copies of an array.

The proposed framework for automatic data layout is currently being implemented as part of the D programming environment. The implementation will provide the basis for the validation of our proposed techniques.

8 Conclusion

We have presented an approach to automatic data layout in the context of a programming tool that produces High Performance Fortran or a similar language as output. This has permitted us to explore exact solutions to the problem of automatic data layout, even though our formulation of the problem is NP-complete. Through the use of the latest and most powerful general purpose techniques for linear and integer programming, we have shown the technique to be practical for a full-sized application.

Recent experiences with similar NP-complete problems indicate that special purpose linear and integer programming techniques can be used to compute the exact solution even faster. These special purpose techniques will take advantage of the particular structure of our formulation of the data layout problem.

Acknowledgement

We would like to thank Hristo Djidjev, Reinhard von Hanxleden, and John Mellor-Crummey for providing many helpful comments.

References

[1] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, second edition, 1986.

[2] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation*, Albuquerque, NM, June 1993.

[3] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. The traveling salesman problem. 1993. In preparation.

[4] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[5] R. Bixby. Implementing the Simplex method: The initial basis. *ORSA Journal on Computing*, 4(3), 1992.

[6] R. Bixby. Progress in linear programming. *ORSA Journal on Computing*, 6(1), 1994.

[7] R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. Technical Report CRPC-TR93-349-S, Center for Research on Parallel Computation, Rice University, November 1993.

[8] B. Chapman, H. Herbeck, and H. Zima. Automatic support for data distribution. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[9] S. Chatterjee, J.R. Gilbert, and R. Schreiber. The alignment-distribution graph. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[10] S. Chatterjee, J.R. Gilbert, R. Schreiber, and S-H. Teng. Automatic array alignment in data-parallel programs. In *Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages*, Albuquerque, NM, January 1993.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[12] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a large scale traveling salesman problem. *Operations Research*, 7:58–66, 1954.

[13] M. Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers.* PhD thesis, University of Illinois at Urbana-Champaign, September 1992.

[14] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[15] D. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[16] K. Kennedy and U. Kremer. Initial framework for automatic data layout in Fortran D: A short update on a case study. Technical Report CRPC-TR93-324-S, Center for Research on Parallel Computation, Rice University, July 1993.

[17] C.W. Keßler. Knowledge-based automatic parallelization by pattern recognition. In C.W. Keßler, editor, *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*, pages 110–135. Verlag Vieweg, Wiesbaden, Germany, 1993.

[18] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, 1990.

[19] K. Knobe, J.D. Lukas, and W.J. Dally. Dynamic alignment on distributed memory systems. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, 1992.

[20] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, MA, 1994.

[21] U. Kremer. NP-completeness of dynamic remapping. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[22] U. Kremer, J. Mellor-Crummey, K. Kennedy, and A. Carle. Automatic data layout for distributed-memory machines in the D programming environment. In C.W. Keßler, editor, *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*, pages 136–152. Verlag Vieweg, Wiesbaden, Germany, 1993.

[23] P. Lee and T-B. Tsai. Compiling efficient programs for tightly-coupled distributed memory computers. In *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993.

[24] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13(4):213–221, August 1991.

[25] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33:60–100, 1991.

[26] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[27] C. Tseng *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, Houston, TX, January 1993. Rice COMP TR93-199.

[28] S. Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1991.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Processor Tagged Descriptors: A Data Structure for Compiling for Distributed-Memory Multicomputers*

Ernesto Su, Daniel J. Palermo, and Prithviraj Banerjee

Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, 1308 West Main Street, Urbana, IL 61801, U.S.A.

Abstract: The computation partitioning, communication analysis, and optimization phases performed during compilation for distributed-memory multicomputers require an efficient way of describing distributed sets of iterations and regions of data. Processor Tagged Descriptors (PTDs) provide these capabilities through a single set representation parameterized by the processor location for each dimension of a virtual mesh. A uniform representation is maintained for every processor in the mesh, whether it is a boundary or an interior node. As a result, operations on the sets are very efficient because the effect on all processors in a dimension can be captured in a single symbolic operation. In addition, PTDs are easily extended to an arbitrary number of dimensions, necessary for describing iteration sets in multiply nested loops as well as sections of multidimensional arrays. Using the symbolic features of PTDs it is also possible to generate code for variable numbers of processors, thereby allowing a compiled program to run unchanged on varying sized machines. The PARADIGM (PARAllelizing compiler for DIstributed-memory General-purpose Multicomputers) project at the University of Illinois utilizes PTDs to provide an automated means to parallelize serial programs for execution on distributed-memory multicomputers.

Keyword Codes: C.1.2; E.2
Keywords: Multiprocessors; Data Storage Representations

1 Introduction

A parallelizing compiler for distributed-memory multicomputers must be able to: (1) describe decompositions of arrays and loops across a number of processors, and (2) translate indices and iterations between global and local address and iteration spaces. There is a need for a representation capable of describing *partitioned* polyhedra of arbitrary dimensions in any of these spaces. It must also provide a flexible structure for performing high-level set operations required in both computation partitioning and communication optimizations [11]. This paper describes how PTDs efficiently provide all of these capabilities. In addition, a key feature of PTDs is the support of code generation for variable numbers of processors. Not only does this allow a compiled program to run unchanged on

*This research was supported in part by the Office of Naval Research (Contract N00014-91J-1096), and in part by the National Aeronautics and Space Administration (Contract NASA NAG 1-613).

varying sized machines, but it is also useful when selecting the number of tasks for multithreaded execution [9] or for varying the number of processors assigned to a given task when utilizing functional parallelism [12]. In contrast, most ongoing work on compilers for distributed memory [3, 5, 8, 10] can only compile for a fixed number of processors.

Earlier structures describing index or iteration space arose in the context of interprocedural data dependence analysis. They include the *Regular Section Descriptor* (RSD) [4] and the *Data Access Descriptor* (DAD) [2]. RSDs can only describe a single row, column, or diagonal of an array, or the entire array. DADs describe convex polyhedra bounded by hyperplanes that are either orthogonal to an axis or at 45° angles with a pair of axes. Both representations are only able to define regions in an unpartitioned address space. The Fortran D compiler [8] uses the RSD augmented with three component sections for each dimension to handle boundary conditions, which are often present in partitions owned by boundary processors. However, when the exact set boundaries are not computable at compile time, it maintains an individual iteration or index set per processor.

The Stanford SUIF compiler system [15] models data and computation decompositions using a form of parametric integer programming. Its representation is parameterized with processor information and is able to handle symbolic block sizes but results in multiversion copies of loops.[1] It also only supports global index and iteration domains, while local index translation is performed at run time for all distributed array accesses [1].

The PTD and defined operations are presented in Section 2. The application of PTDs in the PARADIGM compiler is described in Section 3, and conclusions appear in Section 4.

2 Processor Tagged Descriptors

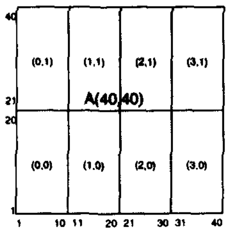
The PTD is a set representation parameterized by processor coordinates. This provides a uniform and efficient means of describing sets of iterations or sections of arrays that are partitioned across processors. A PTD in n dimensions is a collection of one-dimensional PTD$_i$ ($1 \leq i \leq n$), each of which has as a lower and upper bound. Each bound is a symbolic expression involving a processor coordinate (p_i) along dimension i of the mesh:

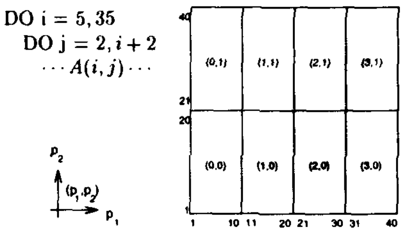
$$
\begin{aligned}
PTD_i &= [lower\_bound : upper\_bound] \\
lower\_bound &= bound \mid \lceil bound \rceil \\
upper\_bound &= bound \mid \lfloor bound \rfloor
\end{aligned}
\qquad
\begin{aligned}
bound &= \begin{cases} linear\_expr(p_i) \\ min(bound, bound) \\ max(bound, bound) \end{cases} \\
linear\_expr(p_i) &= sym\_expr \cdot p_i + sym\_expr
\end{aligned}
$$

where *sym_expr* is a scalar symbolic expression (expressions involving constants and program variables). Each PTD$_i$ captures the range of integers between the lower and upper bounds in a single dimension. Together, they span the entire n-dimensional region. Since the PTD is parameterized by processor coordinates, it provides a uniform representation for every processor in the mesh. This is regardless of the shape and size of the partition and whether it resides in a boundary or an interior processor in the mesh. Furthermore, set operations on these partitions are very efficient because all processors in one or more dimensions can be captured into a single symbolic set operation.

Figure 1 shows the PTD for an array $A(40,40)$ partitioned by blocks on a 4×2 processor mesh. A_1 (first dimension of A) is distributed along the first dimension (4 processors) of the mesh and A_2 along the second dimension (2 processors). Section 3.1 defines this as the IMAGE of A. Each processor is identified by its coordinates (p_1, p_2), with $p_1 \in \{0,1,2,3\}$

[1] For each dimension of a decomposition, SUIF generates three different versions of each partitionable loop (two boundaries and a core computation). For n-dimensions this results in code expansion of 3^n.





```
DO i = 5, 35
  DO j = 2, i + 2
    ··· A(i, j) ···
```

$\mathrm{PTD}_1 = [1 + 10p_1 : 10 + 10p_1]$

$\mathrm{PTD}_2 = [1 + 20p_2 : 20 + 20p_2]$

Figure 1: Partitioned Array $A(40,40)$

$i \in \mathrm{PTD}_1 = [max(5, 1 + 10p_1) : min(35, 10 + 10p_1)]$

$j \in PTD_2 = [max(2, 1 + 20p_2) : min(i + 2, 20 + 20p_2)]$

Figure 2: Partitioned Iteration Space

and $p_2 \in \{0,1\}$. Given any processor $p = (p_1, p_2)$, the bound expressions of the PTD shown in the figure define the regions of A stored in p. Let $A(i,j)$ be referenced in the loop nest as shown above. If the iteration space were partitioned such that p executes iteration (i,j) if p accesses $A(i,j)$ locally, the resulting description would be as shown in Figure 2. Although all of these regions have different sizes (from $\emptyset$ in $p = (0,1)$ to 10×19 in $p = (2,0)$) or shapes (null, rectangular, triangular, and trapezoidal), a single PTD describes them all. Section 3.1 refers to each partition as the ACCESS iteration set of $A(i,j)$ with respect to p.

2.1 Variable Number of Processors

On most multicomputers, users are able to request the size of a machine partition on which to run programs. Given a fixed size array, the more processors available at run time, the smaller the array partition accessed by each processor (since $b_i \propto \frac{1}{P_i}$). The compiled program must adapt itself to the configuration of the machine partition on which it runs. The PTD supports variable processor compilation by retaining the block sizes of distributed array dimensions, b_i, as symbolic variables (described in Section 2.3).

Many implementation issues arise when dynamically allocating memory which has to be accessed in a multidimensional fashion. One solution is to *linearize* all partitioned n-D arrays, as used by APR in the FORGE xHPF parallelizer, at the expense of evaluating complicated array subscripts for each reference at run time. A simpler approach, currently used in PARADIGM, is to require the user to specify a *minimum processor configuration* at compile time. This allows memory to be statically partitioned for the minimum configuration (retaining any dimension information) while allowing the number of processors to grow beyond this lower bound. Since the required block sizes of the partitioned arrays decrease with increasing number of processors, all accesses are contained within the partitioning of the minimum configuration while retaining multidimensional access.

2.2 Set Operations

To work with PTDs, several relational tests are defined to detect subset, disjoint, and null set conditions. Union, intersection, and difference functions are also defined to perform multidimensional operations on the sets. Figure 3 shows algorithms for the PTD set operations. Note that the representation is closed under *intersection* while a *union*

```
is_subset$_{(1d)}(A, B)$
  if $A = \emptyset$          return(true)
  else if $B = \emptyset$   return(false)
  else if $((lb(A) \geq lb(B))$ or $(ub(A) \leq ub(B)))$
    return(true)
  else return(false)
```

(a) Subset Relation ($A_i \subseteq B_i$)

```
are_disjoint$_{(1d)}(A, B)$
  if ($A = \emptyset$ or $B = \emptyset$)
    return(true)
  else if $((ub(A) < ib(B))$ or $(lb(A) > ub(B)))$
    return(true)
  else return(false)
```

(b) Disjoint Relation ($A_i \bigcap B_i = 0$)

```
$(A = \emptyset) \Rightarrow (lb(A_i) > ub(A_i))$   ▷ for some dimension $i$
```

(c) Null Set Relation ($A = \emptyset$)

```
union$(A, B)$
  for $i = 1$ to $dim$ do
    if $A_i = \emptyset$          $A_i \bigcup B_i = B_i$
    else if $B_i = \emptyset$   $A_i \bigcup B_i = A_i$
    else if are_disjoint$_{(1d)}(A_i, B_i)$
      $A_i \bigcup B_i = A_i + B_i$
    else $A_i \bigcup B_i = [\min(lb(A_i), lb(B_i))$ :
                             $\max(ub(A_i), ub(B_i))]$
  done
```

(d) Union Operation ($A \bigcup B$)

```
intersection$(A, B)$
  for $i = 1$ to $dim$ do
    if are_disjoint$_{(1d)}(A_i, B_i)$
      $A_i \bigcap B_i = \emptyset$
    else $A_i \bigcap B_i = [\max(lb(A_i), lb(B_i))$ :
                             $\min(ub(A_i), ub(B_i))]$
  done
```

(e) Intersection Operation ($A \bigcap B$)

```
difference$(A, B)$
  for $i = 1$ to $dim$ do
    if are_disjoint$_{(1d)}(A_i, B_i)$ $A_i - B_i = A_i$
    else
      $T = [lb(A_i) : lb(B_i)-1] \bigcup [ub(B_i)+1 : ub(A_i)]$
      $kernel_i = T \bigcap A_i$
  done
  $A - B =$ stretch$(kernel, A \bigcap B)$
```

(f) Difference Operation ($A - B$)

Figure 3: Algorithms for Performing Set Operations on PTDs

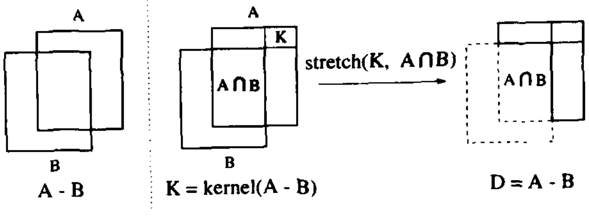
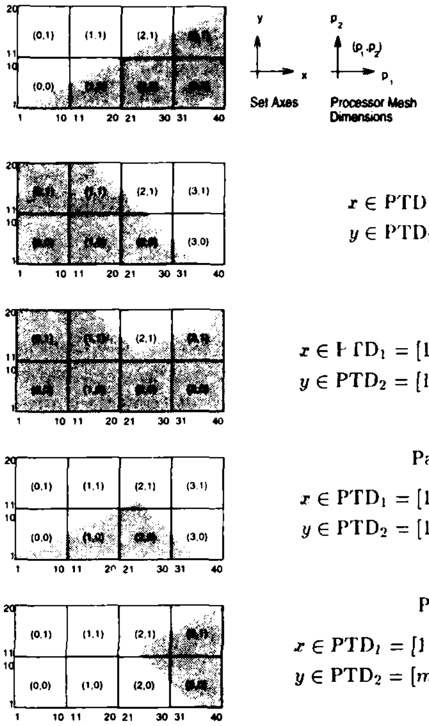


Figure 4: Stretch of a Difference Kernel



Partitioned Set A

$x \in \mathrm{PTD}_1 = [1 + 10p_1 \ : \ 10 + 10p_1]$
$y \in \mathrm{PTD}_2 = [1 + 10p_2 \ : \ min(x/2, 10 + 10p_2)]$

Partitioned Set B

$x \in \mathrm{PTD}_1 = [1 + 10p_1 \ : \ min(35, 10 + 10p_1)]$
$y \in \mathrm{PTD}_2 = [1 + 10p_2 \ : \ min(35 - x, 10 + 10p_2)]$

Partitioned Set Union $A \bigcup B$

$x \in \mathrm{PTD}_1 = [1 + 10p_1 \ : \ 10 + 10p_1]$
$y \in \mathrm{PTD}_2 = [1 + 10p_2 \ : \ min(10 + 10p_2, max(x/2, 35 - x))]$

Partitioned Set Intersection $A \bigcap B$

$x \in \mathrm{PTD}_1 = [1 + 10p_1 \ : \ min(35, 10 + 10p_1)]$
$y \in \mathrm{PTD}_2 = [1 + 10p_2 \ : \ min(min(x/2, 10 + 10p_2), 35 - x)]$

Partitioned Set Difference $A - B$

$x \in \mathrm{PTD}_1 = [1 + 10p_1 \ : \ 10 + 10p_1]$
$y \in \mathrm{PTD}_2 = [min(36 - x, 11 + 10p_2) \ : \ min(x/2, 10 + 10p_2)]$

Figure 5: Examples of Set Operations between two PTDs A and B

or *difference* can potentially result in a list of sets. In Figure 3f, the *difference* operation is seen to perform a *stretch* operation after computing an n-dimensional "kernel". The kernel of a difference can be thought of as the region in which all 1 D differences intersect when projected into n dimensions. The kernel is then *stretched* over the intersection of the original sets. This is done by choosing up to $(n - 1)$ non-null dimensions from one set and selecting the remaining dimensions from the other (see Figure 4).

Figure 5 shows an example of how a union, intersection, and difference are performed between a triangular region A and a trapezoidal region B, both partitioned on a 4 × 2 processor mesh. As shown in the figure, the first axis, x, is aligned with the first processor mesh dimension p_1, and y is aligned with p_2. In the figure, each shaded region corresponds to the accompanying PTD expressions. PTD_1 defines its x-values, and PTD_2 defines the y-values as a function of the x-values (the expressions have been simplified where possible).

2.3 Symbolic Bound Comparison

Symbolic comparison is used to determine relations between bounds of PTDs as well as to maintain simplified bound expressions. The details involved in performing symbolic comparison between the expressions supported in PTDs are not included here due to space constraints. The basic approach taken to perform comparisons between two expressions is based upon a hierarchy of symbolic comparisons at different levels of complexity.

At the lowest level, symbolic comparison is performed between a pair of *symbolic scalar expressions* (expressions involving constants and variables) by simplifying a difference of the two expressions. For *symbolic linear expressions* (a linear expression with symbolic scalar coefficients), linear comparisons can be made within a desired range using scalar comparison between the coefficients. Comparison of *symbolic binary expressions* (a *min* or *max* of two symbolic linear expressions) can be performed with comparison lattices using linear comparisons between components. These levels are applied recursively to perform comparisons of nested binary operations on linear expressions of symbolic quantities.

Symbolic Comparison with Variable Processors Recall that the block sizes of partitioned arrays are allowed to vary with the number of processors P_i assigned to the mesh dimension. Given a dimension of a distributed array of size N, the block size is computed as $b_i = \left\lceil \frac{N}{P_i} \right\rceil$. *Since all array dimensions mapped to the same mesh dimension* will vary together (with respect to P_i), comparison relations still exist between symbolic block sizes although the actual value of the block size is not known at compile time.

To support comparison of symbolic block sizes the *symbolic scalar comparison* operation is further abstracted using *mirrors*. A *mirror* retains information that would otherwise be lost if the source of the symbolic quantity were ignored. For symbolic scalars, an n-dimensional polynomial is used to *mirror* the value involving symbolic block sizes:

$$\beta_n P_n^{-1} + \beta_{n-1} P_{n-1}^{-1} + \ldots + \beta_2 P_2^{-1} + \beta_1 P_1^{-1} + c$$

where β_i are block sizes based on a minimum processor configuration and c is a constant. Whenever symbolic scalars are encountered, comparison can be performed by componentwise comparisons of each of the corresponding coefficients of the mirrors maintained for each scalar.[2] Any arithmetic operation that is performed on a symbolic expression must also be performed on the corresponding mirror polynomial. For use in the compiler, it is sufficient to support polynomial addition/subtraction, and scalar multiplication/division on these mirrors.

3 Use of PTDs in Compilation

Both data and computation decompositions can be represented using PTDs. In this section we examine the use of PTDs in the PARADIGM compiler [7]. During analysis of the reference patterns in a program, PTDs are transformed among different domains. Altogether, four domains can be defined: global and local indices (**GN** and **LN**), and global and local iterations (**GI** and **LI**). Transformation functions (and their inverses) among the different domains are:

$$\begin{tikzcd}
& \text{Global} & & \text{Local} \\
\text{Indices} & \mathbf{GN} \arrow[rr, "\tau_p(i)"] \arrow[d, shift right, "s^{-1}(i)"'] & & \mathbf{LN} \\
\text{Iterations} & \mathbf{GI} \arrow[u, shift right, "s(i)"'] \arrow[rr, "\lambda_p(i)"'] & & \mathbf{LI} \arrow[u, "s(i)"']
\end{tikzcd}$$

[2] Since any dimension of a mirror is affected by the constant term, c is added to non-zero coefficients during comparisons. Constant scalars also have implicit mirrors where c is equal to the numeric value.

The subscript function $s(i)$ of an array reference $A(s(i))$, enclosed in a loop, transforms points from the loop's iteration space into the index space.[3] For simplicity, in this paper $s(i)$ is allowed to be a linear function[4] of a loop variable i: $s(i) = a_1 i + a_0$. The index translation function τ_p transforms points in the global index space into indices local to a processor p: $\tau_p(x) = x - bp$, where b is the block size of distribution. Similarly, the iteration translation function λ_p transforms points from the global iteration space into iterations local to a processor p: $\lambda_p(i) = i - \frac{b}{a_1}p$.

3.1 Data Partitioning

IMAGE Global Index Sets Using the data partitioning provided by the user or generated by the compiler [6], PTDs in the global index domain (**GN**) can be generated for each partitioned dimension of a distributed array. For each array $A(1 : N)$ distributed across a processor mesh dimension of P processors indexed by p $(0 \leq p \leq P - 1)$, the corresponding PTD describing this partitioned array is

$$\text{PTD} = [1 + bp : b + bp]$$

where $b = \left\lceil \frac{N}{P} \right\rceil$ is the *block size* of A. This PTD is called the IMAGE of A on p, denoted IMAGE(A, p). This expression assumes that the alignment offset ω of A on the mesh dimension is $\omega = 0$, and that the indices of A start at 1. In general, we may have $A(\alpha : N + \alpha - 1)$ and $\omega \neq 0$, and then IMAGE(A, p) becomes

$$\text{IMAGE}(A, p) = [max(\alpha, \alpha + \omega + bp) : min(N + \alpha - 1, b + \omega + bp)]$$

If $\omega > 0$, then we need some $p < 0$ to address the leftmost ω elements of A. Similarly, if $\omega < 0$, then some $p > P - 1$ is required to address the rightmost ω elements of A. In either case, we can define $p_i' = p \ mod \ P$ as the real processor coordinate. To avoid unnecessarily obscuring the expressions derived, we assume $\omega = 0$ and $\alpha = 1$.

ITERIMAGE and ACCESS Global Iteration Sets From the IMAGE index set of A, the ITERIMAGE and ACCESS global iteration sets (in **GI**) of each reference $A(s(i))$ can be computed. The ITERIMAGE set of $A(s(i))$ with respect to a processor p is the set of iterations i, unrestricted by loop bounds, such that $A(s(i))$ is stored in p. Since the partition of A stored in p is given by IMAGE(A, p), we can write

$$\begin{aligned} \text{ITERIMAGE}(A(s(i)), p) &= \{i \mid s(i) \in \text{IMAGE}(A, p)\} \\ &= [\lceil s^{-1}(1 + bp) \rceil : \lfloor s^{-1}(b + bp) \rfloor] \end{aligned}$$

For $A(s(i))$ enclosed in a loop nest in which the i-loop goes from l to u, ACCESS$(A(s(i)), p)$ is defined as the set of iterations i in the i-loop for which $A(s(i))$ is stored in p:

$$\begin{aligned} \text{ACCESS}(A(s(i)), p) &= \text{ITERIMAGE}(A(s(i)), p) \bigcap [l : u] \\ &= [\lceil max(l, s^{-1}(1 + bp)) \rceil : \lfloor min(u, s^{-1}(b + bp)) \rfloor] \end{aligned}$$

[3] For simplicity of notation, it suffices to use one-dimensional arrays in the discussion. For n dimensions, there is a PTD$_i$ per dimension. Thus, $A(s(i))$ should be thought of as $A(\ldots, s(i), \ldots)$.

[4] The more general case of affine subscript functions over enclosing loop variables can also be handled. If i is the innermost loop variable that appears in the subscript, then $s(i) = a_1 i + a_0$ still holds by letting a_0 be an affine function over the rest of the loop variables. Some additional conditions for loop partitioning must be observed when such subscripts are allowed.

3.2 Computation Partitioning

The compiler applies the *owner computes rule* which states that the processor owning a data item performs all computations for that item. An efficient implementation must avoid the run-time overhead of computing ownership. For a loop nest, this is done by reducing the loop bounds according to the Reduced Iteration Set [8] (RIS) of the loop. This is the union of the ACCESS sets of the left-hand side (*lhs*) references in the loop with respect to a processor p. The RIS represents the largest subset of the iteration space for which p does some work in *every* iteration. The RIS is in the global iteration space **GI**. If used directly, an index translation τ_p is required at every iteration to transform $s(i)$ from the global index space **GN** into the local space **LN** of p: DO ($i \in$ RIS) A(τ_p(s(i)))=$\cdots$ This can be costly if |RIS| is large. A better approach is to obtain LRIS = λ_p(RIS), which is in the local iteration space **LI** of p. The loop becomes DO ($i \in$ LRIS) A(s(i))=$\cdots$ and is more efficient since |RIS| τ_p operations are saved at run time.

If the ACCESS set of a *lhs* in the loop is a *proper* subset of the RIS, then the corresponding statement must be *masked* [8] to make its execution conditional. Optimizations such as *mask merging* (consecutive assignment statements sharing the same mask) and *mask extraction* (multiple occurrences of a mask coalesced and extracted out of a loop) are performed automatically by the compiler. Details about the conditions and algorithms to compute RIS, LRIS, and masks can be found in [13].

3.3 Communication Determination

To find the necessary communication among processors, the compiler computes several *communication sets* for each right-hand side (*rhs*) reference. Details of these sets and the algorithms to calculate them are found in [13, 14]. A brief description follows.

RECEIVE and SEND Global Iteration Sets The first communication sets computed are the RECEIVE and SEND sets, both in the global iteration space **GI**. For an assignment statement S with a *lhs* reference L and a *rhs* reference R, RECEIVE(R, p) is the set of global iterations for which p owns L but not R. Thus, p must *receive* this data via interprocessor communication to execute S. Similarly, SEND(R, p) is the set of global iterations for which p owns R but not L, and hence must *send* the data to the owner processor of L for it to execute S. Thus, we have:

$$\begin{aligned} \text{RECEIVE}(R,p) &= \text{ACCESS}(L,p) - \text{ITERIMAGE}(R,p) \\ \text{SEND}(R,p) &= \text{ACCESS}(R,p) - \text{ITERIMAGE}(L,p) \end{aligned}$$

IN and OUT Local Index Sets To generate communication, the array sections to be communicated must be determined. The RECEIVE and SEND sets are translated from the global iteration space **GI** to the local index space **LN** of each processor p, obtaining the IN and OUT sets, respectively. First the subscript function $s(i)$ is applied to translate the sets from global iterations **GI** to global indices **GN**, and then the index translation function τ_p is used to take the sets from global indices **GN** to local indices **LN** of p:

$$\begin{aligned} \text{IN}(R,p) &= \tau_p(s(\text{RECEIVE}(R,p))) \\ \text{OUT}(R,p) &= \tau_p(s(\text{SEND}(R,p))) \end{aligned}$$

PARADIGM Code Example It is also of interest to examine the actual code generated. Figure 6 shows a serial program for a two-dimensional Jacobi iteration. Figure 7 shows the resulting optimized node program supporting a variable number of iPSC processors with a minimum configuration of four nodes. Notice the use of symbolic block sizes (highlighted) whose values are computed at *run time*.

```
program jacobi
parameter (np2 = 500, ncycles = 10)
real A(np2, np2), B(np2, np2)
np1 = np2-1
do k = 1, ncycles
    do j = 2, np1
        do i = 2, np1
            A(i, j) = (B(i-1, j) + B(i+1, j)
                + B(i, j-1) + B(i, j+1)) / 4
        end do
    end do
    do j = 2, np1
        do i = 2, np1
            B(i, j) = A(i, j)
        end do
    end do
end do
end
```

Figure 6: Serial Version of Jacobi

```
program jacobi
character m$buf(1000)
integer my$p(2), m$bA1, m$bA2, m$bB1, m$bB2
integer m$numdim, m$num(2), m$to(-1:1,-1:1), m$inc
real A(250,250), B(0:251,0:251)
m$numdim = 2     {number of mesh dimensions}
m$num(1) = 2     {minimum mesh configuration}
m$num(2) = 2
call m$getnum(m$num,numnodes())
call m$gridinit(m$numdim,m$num,numnodes(),1)
call m$gridcoord(mynode(),my$p)
m$to(-1,0) = m$gridrel2(my$p,-1,0)
m$to(0,-1) = m$gridrel2(my$p,0,-1)
m$to(0,1) = m$gridrel2(my$p,0,1)
m$to(1,0) = m$gridrel2(my$p,1,0)
m$bA1 = ceil(float(500) / m$num(1))
m$bA2 = ceil(float(500) / m$num(2))
m$bB1 = ceil(float(500) / m$num(1))
m$bB2 = ceil(float(500) / m$num(2))
do k = 1,10
   if (my$p(2) .ge. 1) then
      call csend(0,B(1,1),4*m$bB1,m$to(0,-1),1)
   end if
   if (my$p(2) .le. m$num(2)-2) then
      call crecv(0,B(1,m$bB2+1),4*m$bB1)
   end if
   if (my$p(2) .le. m$num(2)-2) then
      call csend(1,B(1,m$bA2),4*m$bB1,m$to(0,1),1)
   end if
   if (my$p(2) .ge. 1) then
      call crecv(1,B(1,0),4*m$bB1)
   end if
   if (my$p(1) .ge. 1) then
      m$inc = f$pack2(B,4,0,251,0,251,1,1,1,1,m$bB2,1,m$buf)
      call csend(2,m$buf,m$inc,m$to(-1,0),1)
   end if
   if (my$p(1) .le. m$num(1)-2) then
      call crecv(2,m$buf,1000)
      m$inc = f$unpack2(B,4,0,251,0,251,m$bB1+1,m$bA1+1,
                       1,1,m$bB2,1,m$buf)
   end if
   if (my$p(1) .le. m$num(1)-2) then
      m$inc = f$pack2(B,4,0,251,0,251,m$bA1,m$bB1,
                       1,1,m$bB2,1,m$buf)
      call csend(3,m$buf,m$inc,m$to(1,0),1)
   end if
   if (my$p(1) .ge. 1) then
      call crecv(3,m$buf,1000)
      m$inc = f$unpack2(B,4,0,251,0,251,0,0,1,1,m$bB2,1,m$buf)
   end if
   do j = max(2-m$bA2*my$p(2), 1),
            min(499-m$bA2*my$p(2), m$bA2)
      do i = max(2-m$bA1*my$p(1), 1),
               min(499-m$bA1*my$p(1), m$bA1)
         A(i, j) = (B(i-1, j) + B(i+1, j) + B(i, j-1)
                                + B(i, j+1)) / 4
      end do
   end do
   do j = max(2-m$bB2*my$p(2), 1),
            min(499-m$bB2*my$p(2), m$bB2)
      do i = max(2-m$bB1*my$p(1), 1)),
               min(499-m$bB1*my$p(1), m$bB1)
         B(i, j) = A(i, j)
      end do
   end do
end do
end
```

Figure 7: Jacobi for Variable Number of Processors, with a Minimum 2 × 2 Configuration

4 Conclusions

The purpose of this work is to provide a uniform and efficient way of describing distributed data or computation. The main advantages of the PTD over many other existing representations can be summarized as follows: (1) a single PTD describes all partitions of a region regardless of the shape and size of each partition; (2) a wide range regions of are possible with arbitrary numbers of dimensions, arbitrary boundary angles (not just multiples of 45°), and nonconvex shapes; (3) the PTD supports symbolic set functions and domain transformations, essential parallelizing compiler for distributed memory; and (4) symbolic block sizes make it possible to compile for a variable number of processors.

Acknowledgements: The authors would like to thank the reviewers for their helpful input. We would also like to thank Christy Palermo for her help and suggestions with the development of the symbolic comparison algorithms.

References

[1] S. P. Amarasinghe and M. S. Lam. "Communication Optimization and Code Generation for Distributed Memory Machines," in *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 126–138, June 1993.

[2] V. Balasundaram. "A Mechanism for Keeping Useful Internal Information in Parallel Programming Tools: the Data Access Descriptor," *Journal of Parallel and Distributed Computing*, 9(2):154–170, June 1990.

[3] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. "Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers, Design, Implementation, and Performance Results," *Proceedings of the 1993 ACM International Conference on Supercomputing*, pages 351–360, July 1993.

[4] D. Callahan and K. Kennedy. "Analysis of interprocedural side effects in a parallel programming environment," in *Proceedings of the First ACM International Conference on Supercomputing*, pages 138–171, Athens, Greece, 1987.

[5] B. Chapman, P. Mehrotra, and H. Zima. "Programming in Vienna Fortran," in *Third Workshop on Compilers for Parallel Computers*, pages 145–164, 1992.

[6] M. Gupta and P. Banerjee. "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[7] M. Gupta and P. Banerjee. "PARADIGM: A Compiler for Automated Data Partitioning on Multicomputers," in *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[8] S. Hiranandani, K. Kennedy, and C. Tseng. "Compiling Fortran D for MIMD Distributed Memory Machines," *Communications of the ACM*, 35(8):66–80, August 1992.

[9] J. G. Holm, A. Lain, and P. Banerjee. "Compilation of Scientific Programs into Multithreaded and Message Driven Computation," to appear in the *1994 Scalable High Performance Computing Conference*, 1994.

[10] C. Koelbel. "Compile-Time Generation of Regular Communications Patterns," in *Proceedings of Supercomputing '91*, pages 101–110, Albuquerque, NM, November 1991.

[11] D. J. Palermo, E. Su, J. A. Chandy, and P. Banerjee. "Compiler Optimizations for Distributed Memory Multicomputers used in the PARADIGM Compiler," to appear in the *1994 International Conference on Parallel Processing*, August 1994.

[12] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. "A Convex Programming Approach for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers," to appear in the *1994 International Conference on Parallel Processing*, 1994.

[13] E. Su. "Automating Parallelization of Regular Computations for Distributed-Memory Multicomputers," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1993.

[14] E. Su, D. J. Palermo, and P. Banerjee. "Automating Parallelization of Regular Computations for Distributed Memory Multicomputers in the PARADIGM Compiler," in *Proceedings of the 1993 International Conference on Parallel Processing*, pages II:30–38, August 1993.

[15] S. Tjiang, M. Wolf, M. Lam, K. Pieper, and J. Hennessy. "Integrating scalar optimizations and parallelization," in U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua editors, *Languages and Compilers for Parallel Computing*, pages 137–151, Berlin, Germany, 1992. Springer-Verlag.

PART V

MULTI-LEVEL PARALLELISM

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Resource Spackling: A Framework for Integrating Register Allocation in Local and Global Schedulers[†]

David A. Berson, Rajiv Gupta, and Mary Lou Soffa

Computer Science Department, University of Pittsburgh, Pittsburgh, Pa. 15260

Keyword Codes: D.1.3; C.1.3
Keywords: Instruction-level parallelism; Speculative code motion; VLIW, Superscalar

Abstract: We present Resource Spackling, a framework for integrating register allocation and instruction scheduling that is based on a *Measure* and *Reduce* paradigm. The technique measures the resource requirements of a program and uses the measurements to distribute code for better resource allocation. The technique is applicable to the allocation of different types of resources. A program's resource requirements for both register and functional unit resources are first measured using a unified representation. These measurements are used to find areas where resources are either under or over utilized, called *resource holes* and *excessive sets*, respectively. Conditions are determined for increasing resource utilization in the resource holes. These conditions are applicable to both local and global code motion.

1 Introduction

A variety of local and global scheduling techniques have been developed for exploiting instruction level parallelism. The degree of parallelism in the schedule is affected by register allocation, which is applied either before or after scheduling. Instruction scheduling, which allocates functional units, and register allocation have competing goals. The goal of register allocation is to avoid spills, which tends to result in a few values being held in registers for a long time. The goal of instruction scheduling is to keep all functional units busy, typically requiring a large number of values to be available for future operations. Thus an improvement in the availability of one resource may reduce the availability of the other resource, and possibly result in poor overall quality of generated code.

We present a framework for integrating instruction scheduling and register allocation that is based upon a *Measure* and *Reduce* paradigm. Resource requirements are measured and better resource utilization in both local and global scheduling is achieved by moving instructions from areas with over utilized resources to areas with under utilized resources. Integrated allocation of registers and functional units is achieved by allowing simultaneous consideration of the demand for both types of resources during scheduling.

Previous work on *local* schedulers has treated register allocation and instruction scheduling as separate phases [BEH91, GoH88, Pin93, SwB90]. In addition, the instruction scheduling phases have been based on list scheduling. The separation of phases and use of list scheduling limit the direct assessment of the impact of register and functional unit

[†] Partially supported by National Science Foundation Presidential Young Investigator Award CCR-9157371 and Grant CCR-91090809 to the University of Pittsburgh

allocation decisions on the availability of other resources and hence on the length of the schedule. Recent work has incorporated parallel live range information into register allocation, but still separates register allocation from instruction scheduling [Pin93]. The approach we present unifies the allocation of registers and functional units in a single phase and is able to consider the impact of an allocation decision on other instructions.

Work on *global* scheduling has identified blocks to which an instruction can be moved [AiN88, BeR91, Fis81, GuS90, SHL92] by concentrating on functional unit constraints [EbN89, MGS92] in carrying out the code motion. Only recently has work on global scheduling begun to consider register allocation as a part of the problem [MoE92, NiG93]. Although Moon and Ebcioglu[MoE92] added register constraints, they are only able to exploit unused registers at the beginning or end of a basic block. Our framework allows all idle resources in a basic block to be identified, regardless of when they are idle, and exploited when instructions are available. The framework can be used in conjunction with commonly used methods for performing code motion, such as Trace Scheduling[Fis81], Percolation Scheduling[AiN88], and Region Scheduling[GuS90].

The Resource Spackling framework computes resource requirements for a program. A unified representation of functional unit and register uses is constructed to identify all resource uses that can temporally share the same instance of a resource. Using this representation we compute the maximum number of each resource required at each point in the program. Maximum functional unit requirements correspond to maximum parallelism, while maximum register requirements correspond to the maximum number of simultaneously live values. The resource requirements measures are then used to identify two sets: *excessive sets*, which are sets of instructions that may be executed concurrently but require more resources than are available, and *resource holes*, which are areas where a resource is underutilized. Properties are identified for holes and resources, which indicate how instructions can be inserted into the holes to increase resource utilization. Moving an instruction is only beneficial if it can be placed in a hole where all necessary resources are available. The technique is called Resource Spackling due to the process of identifying and filling resource holes.

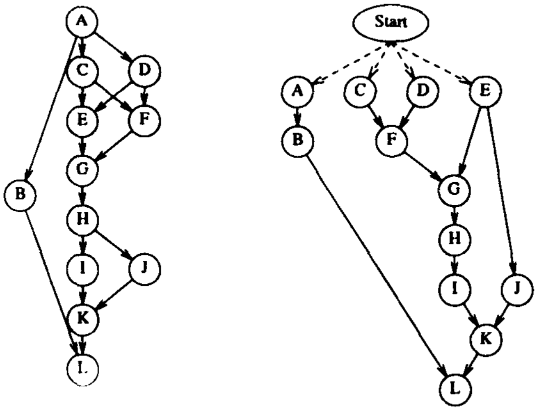
2 Determining Resource Requirements using Allocation Chains

This section summarizes the measurement of resource requirements used to locate excessive sets and resource holes [BGS93]. Measurement of resource demands depends on the usage characteristics of the resource. The two major types of resources considered in this work, functional units and registers, have different use properties. If a resource is in use only during the execution of an instruction, we say it is a *non-spanning* resource. If the use of a resource begins during the execution of one instruction and ends during the execution of a subsequent instruction we say the resource is *spanning.* The instruction that begins the use is called the *defining instruction* and the instruction that ends the use is called the *killing instruction.* Functional units are non-spanning resources, while registers are spanning resources.

The resource usage information is represented as a $Reuse_R$ DAG, and is computed from the program DAG. Functional unit and register *Reuse* DAGs are denoted as $Reuse_{FU}$ DAG and $Reuse_{Reg}$ DAG respectively. The term *Reuse* comes from the property that for any edge `(A, B)` in the $Reuse_R$ DAG, where `A` and `B` are nodes representing instructions in the program DAG, `B` can always safely reuse `A`'s instance of R.

The program DAG in Figure 1(b) is a $Reuse_{FU}$ DAG. A corresponding $Reuse_{Reg}$ DAG is shown in Figure 1(c). The selection of which use kills a value must be performed carefully so that the number of simultaneously live values is maximized. To maximize the number of simultaneously live values for the $Reuse_{Reg}$ DAG in Figure 1(c) the following

```
A: load a
B: b = 2 * a
C: c = a + 1
D: d = a - 3
E: e = c * d
F: f = c - d
G: g = e / f
H: h = g + 5
I: i = h * 2
J: j = h + 4
K: k = i / j
L: l = b + k
```



(a) Basic block of code (b) Functional Unit Reuse DAG (c) Register Reuse DAG

Figure 1: Basic block and *Reuse* DAGs

selections have been made: B kills A, I kills H, and E kills both C and D.

Resource requirements for resource R are measured from the partial order represented by the $Reuse_R$ DAG by finding sets of instructions that can reuse the same resource instance, called *allocation chains.* Formally, a chain is a subset of elements in a partial order such that all elements in the chain are related, i.e., ordered. A decomposition of a partial order into chains is a set of chains such that all elements in the partial order are in exactly one of the chains. A decomposition is minimal if there is no other decomposition of the partial order that has fewer chains. Since the allocation chains are found on the $Reuse_R$ DAG, all instructions on one allocation chain can be assigned the same instance of the resource. Each of the sets of nodes {A, C, E, G, H, I, K, L}, {D, F, J}, and {B} is a chain, and this set of chains forms a minimal decomposition of the DAG. Similarly, the $Reuse_{Reg}$ DAG can be minimally decomposed into the chains {A, B, L}, {C, F, G, H, I, K}, {D}, and {E, J}.

The maximum number of independent elements in a partial order is equal to the number of chains in a minimal decomposition[Dil50]. Since the partial order is constructed from resource reuse information, the number of chains in a minimal decomposition represents the maximum number of instructions that can execute concurrently and simultaneously live values for $Reuse_{FU}$ DAGs and $Reuse_{Reg}$ DAGs respectively. Thus, the block in Figure 1(a) requires three functional units and four registers to exploit all of its parallelism. A minimum decomposition of a partial order can be found by using a straightfor vard transformation to a bipartite graph matching problem[FoF65].

3 Resource Holes

Resource holes and their properties are located by analyzing the allocation chains for the resource of interest. Given a hole h, the size of h, $size_h$, is the number of cycles for which the hole's resource is available for allocation. EAT_h, the *earliest available time* of hole h, is the earliest time that the resource can be allocated to an instruction. LAT_h, the *latest available time* of hole h, is the lastest time that the resource can be allocated to an instruction. These properties are determined by the time of execution of the instructions surrounding the holes. The scheduling of instruction i in the program DAG is limited by the precedence constraints to a time frame in which it can execute. The time frame is delimited by the instruction's earliest start time, EST_i, and latest finish time, LFT_i. Let τ_i denote the execution time for instruction i. Then i's latest start time, LST_i, is given by $LST_i = LFT_i - \tau_i$. The *slack time* for scheduling instruction i is given by $slack_i = LST_i - EST_i$. The identification of resource holes is performed by examining the instructions' ESTs and LFTs on each allocation chain and recording the information for each hole.

Resource holes can occur in two different situations. The first, a free hole, occurs when an instance of a resource is unused in a section of a basic block. Free holes can occur because no instructions from an allocation chain can execute in this section. They can also occur at the beginning or end of a block before maximum demands are encountered.

Definition 1 If two consecutive instructions, i_j and i_{j+1}, on an allocation chain cannot be executed consecutively, i.e., $LFT_{i_j} < EST_{i_{j+1}}$, then there is a *free hole*, h, such that $EAT_h = LFT_{i_j}$, $LAT_h = EST_{i_{j+1}}$, and $size_h = LAT_h - EAT_h$.

We refer to the pair (EAT_h, LAT_h) as the *range* of hole h. As an example, assume the DAG in Figure 1(b) and the chains mentioned earlier, and that all instructions require unit time. The DAG requires eight time units to execute. A free functional unit hole exists between instructions F and J, with size 2 and range (3, 5). Thus, two instructions could be allocated to that allocation chain between F and J.

The second type of hole, a slack hole, occurs when resources that are already allocated may be temporally shared. If $slack_i = 0$ then i is on a critical path and has no flexibility for scheduling without increasing the execution time of the basic block. If i is not on a critical path then there is some flexibility on when it can be scheduled. Thus, its resources may be available for allocation to another instruction.

Definition 2 If there is a set of consecutive instructions $\mathcal{I} = \{i_1, i_2, ..., i_n\}$ and a constant s such that $\forall_{i_j \in \mathcal{I}}\ slack_{i_j} = s$, there is a *slack hole*, h, such that $EAT_h = EST_{i_1}$, $LAT_h = LFT_{i_n}$, and $size_h = s$.

In Figure 1(b) there is a slack functional unit hole involving instructions A, B, and the end of the block. B has a slack time of 5, and a range of (1, 7). Thus, five instructions could be allocated to B's allocation chain. Any number of these five instructions can be allocated between A and B, with the remainder after B.

Instructions are inserted in holes to avoid increasing the critical path length. Thus a hole must be at least as large as the instructions being inserted in it. There are cases when additional instructions must be placed in the hole to manage the use of registers. In a register slack hole there are instructions which are already using the register. Spill code must be placed in the hole around the inserted instructions to free this register for the inserted instructions. Additionally, the final value computed by the inserted instructions must sometimes be spilled. The following theorem formalizes the conditions under which a set of instructions can be inserted in a hole. The values τ_{store} and τ_{load} are the number of cycles required to store and load a value respectively.

Theorem 1 Let $\mathcal{I}$ be a set of instructions, with execution time $\tau_{\mathcal{I}}$, that requires resource R. During insertion of $\mathcal{I}$ in h there will not be any increase in the length of the critical path of the basic block containing h, due to unavailability of resource R, if h satisfies one of the following conditions.

1. R is a non-spanning resource or R is a spanning resource and inserting $\mathcal{I}$ in h requires no spills, and $size_h \geq \tau_{\mathcal{I}}$.
2. R is a spanning resource, and inserting $\mathcal{I}$ in h requires only the final value computed by $\mathcal{I}$ to be spilled and $size_h \geq \tau_{\mathcal{I}} + \tau_{store}$.

3. R is a spanning resource, and inserting $\mathcal{I}$ in h requires only a value computed by instructions already in h to be spilled, and $size_h \geq \tau_{\mathcal{I}} + \tau_{store} + \tau_{load}$.
4. R is a spanning resource, inserting $\mathcal{I}$ in h requires both a value computed by instructions already in h and the final value computed by $\mathcal{I}$ to be spilled and $size_h \geq \tau_{\mathcal{I}} + 2\tau_{store} + \tau_{load}$.

Proof: Since the cases are mutually exclusive, each case is proven in turn.

Case 1 When no spilling is required the hole must be at least as big as the inserted instructions, whose size is $\tau_{\mathcal{I}}$, giving $size_h \geq \tau_{\mathcal{I}}$.

Case 2 When the final value computed by the inserted instructions must be spilled, the hole must be at least large enough to hold both the inserted instructions and the store instruction, giving $size_h \geq \tau_{\mathcal{I}} + \tau_{store}$.

Case 3 When a value computed by instructions already in the hole must be spilled, the value must be stored before the inserted instructions and then reloaded following the inserted instructions. Thus the hole must be at least large enough to hold a store and a load in addition to the inserted instructions, giving $size_h \geq \tau_{\mathcal{I}} + \tau_{store} + \tau_{load}$.

Case 4 This case is a combination of cases 2 and 3. Summing the additional instructions that must be inserted with $\mathcal{I}$ gives $size_h \geq \tau_{\mathcal{I}} + 2\tau_{store} + \tau_{load}$.□

In addition to the instructions chosen for insertion in the hole, $\mathcal{I}$ must also contain any load instructions for any values needed by $\mathcal{I}$ which are not already in registers.

In some situations, instructions must be inserted even when there are no holes available for insertion. In these situations the scheduler creates pseudo holes in the block for each resource needed, resulting in an increase in the critical path length. This process of forcing holes into the block, increasing its critical path length, is called *wedged insertion*.

4 Local Scheduling and Register Allocation

In the *Measure and Reduce* paradigm, local resource allocation is performed by introducing sequentiality between instructions whose resource demands exceed available resources. The sequencing places two instructions, which were on separate allocation chains, onto a single allocation chain. The result is that the two instructions are allocated a single instance of the resource and share it temporally. Sequencing must be performed when the number of allocation chains is greater than the number of resource instances available.

Definition 3 An *excessive set* $E_R = \{I_1, I_2, ... I_m\}$ for resource R is a set of instructions such that

1. all instructions are independent, i.e., $\underset{i,j \in E_R}{\forall} \; i \notin ancestors(j) \cup descendants(j)$, and
2. there are excessive requirements, i.e., $m > |R|$.

Note that condition 1 implies that each instruction is on a separate allocation chain.

For every instruction i, the allocation chains can be used to find all resources for which i is a member of at least one excessive set of the resource. Sequentialization is then performed by selecting i to be the instruction with excessive uses that has the greatest slack time to be moved to holes. The slack time is used to prioritize the instructions since it indicates flexibility in finding a place to move the instruction. If there is a set of overlapping holes for all resources that i excessively uses within i's execution range, then i can be inserted in those holes without increasing the critical path length.

If there is no set of holes within i's execution range, then an increase in the critical path length is unavoidable. There are two options. First, there may be a set of holes close to i's execution range to which i can be moved. Second, wedged insertion can be performed to create a set of holes for i's excessive uses. The option that minimizes the increase to the critical path length should be selected. The outline for an algorithm that reduces a block by finding or creating holes is given in Figure 2.

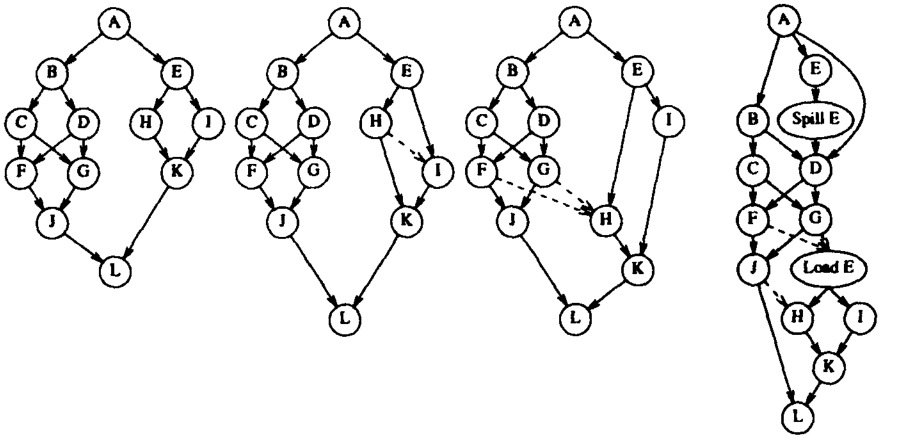
```
Procedure reduce_block( $block$ )
{ While $block$ has excessive sets do
   { $\mathcal{I}$ = all instructions in all excessive sets for all resources;
     select $i \in \mathcal{I}$ with maximum $slack_i$;
     $\mathcal{R}$ = the set of resources that $i$ excessively uses;
     if ($\exists \forall_{r \in \mathcal{R}}$ hole $h_r$ whose ranges overlap with each other and
           $i$'s execution range)
        $holes$ = this set of holes;
     else
     { $close$ = the set of holes $h_r$ s.t. $r \in \mathcal{R}$ whose ranges overlap
           and are closest to $i$'s execution range;
        $wedge$ = the set of holes created by $wedged\ insertion$ for $\mathcal{R}$;
        $holes$ = the set, either $close$ or $wedge$ that minimally increases
           the critical path length of $block$; }
     $\forall_{h_r \in holes}$ place $i$ in $h_r$ by adding $sequentialization$ edges;
     if ( excessive spanning uses remain )
        $spill$ uses between the excessive set and the hole containing $i$;
     remove $i$ from excessive set information; }
}
```

Figure 2: Function reduce_block()

As an example, consider the DAG in Figure 3(a). First assume that the target architecture has at least five registers but only three functional units. Then the nodes C, D, F, G, H, and I are all members of at least one functional unit excessive set. Nodes H and I each have a slack time of one. There is a functional unit slack hole around each of H and I, so H's hole overlaps with I's execution range. Figure 3(b) shows the result of inserting I in H's hole. Dashed arrows indicate sequentializing dependences, i.e., dependences due to reuse of resources rather than data values.

Now assume that only four registers are available and G is selected to kill both C's and D's values and I is selected to kill E's value. Then nodes C, D, F, H, and I are in functional unit and register excessive sets. Node H has slack time but there are no register holes in its execution range. Therefore the algorithm must increase the critical path length. There are a functional unit and a register hole available after G executes since it kills two values and only needs one register for itself. Inserting H in the hole following G would increase the critical path by one instruction. Wedged insertion would increase the critical path length more because the pseudo hole must be large enough to spill and reload a value. Therefore the algorithm chooses the hole close to H instead of performing wedged insertion. The resulting DAG is shown in Figure 3(c).

Although the creation of some live values may be delayed by sequencing, the instructions that compute the live values may need input values. These input values remain live from where they are computed to where the excessive instructions are moved. In Figure 3(c) the value computed by H was delayed until there was a register available. However, E's value remains live until after both H and I execute. In this example it is impossible to reduce the register requirements below four using just sequentialization. When such a situation occurs sequentialization must be combined with register spilling. There are two options for selecting what values to spill. Either the values in the excessive set may be



(a) Example DAG (b) Functional unit sequencing (c) Register sequencing (d) Register spilling

Figure 3: Local reductions of resource requirements

computed and spilled, or the input values may be spilled. The option selected depends on what holes are available. Computing and spilling the excess values prior to the excessive set requires additional functional unit and register holes, while spilling the input values requires additional functional unit holes to where the values are moved.

Continuing with the above example, assume the same killing instructions and that the target architecture has three registers and two functional units. As before, the nodes C, D, F, H, and I are in an excessive set and only instructions H and I have slack time. Free functional unit and register holes become available after G executes, and another set of free functional unit and register holes become available after J executes. H and I are placed in the free holes. However, the sequentialization would still leave the excessive set {C, D, E, F}. Thus a spill must be performed. To minimize the number of spills, the algorithm spills the input value, E. The resulting DAG is shown in Figure 3(d).

5 Global Scheduling and Register Allocation

The goal of global scheduling is to move instructions from a *source* block to a *destination* block to decrease the execution time of the source block by reducing the critical path length in the source block and avoiding increasing the critical path length in the destination block. The instructions moved are called *fill instructions* since they are inserted in holes in the destination block. Fill instructions may be found in blocks with the same control dependences and in blocks with different control conditions when the architecture supports speculative execution [SHL92] or guarded execution [HsD86], or when code duplication is performed.

Next we describe how existing global code motion techniques [AiN88, Fis81, GuS90] can use the framework to unify functional unit and register allocation, and determine which code motions are beneficial. To realize a benefit, all instructions that are at one end of a DAG and are on a critical path must be moved together; otherwise the critical path length will not be reduced. We call such sets of instructions *critical sets*. Consider removing nodes from the top of the DAG in Figure 1(b). The first critical set is {A}. When A is moved the length of the DAG is reduced by the execution time of A. Then the

```
Function fill( $dest$, $source$ )
{ $reduce$ = 0;
   While $dest$ has holes do
   { $cs$ = next set of critical instructions from $source$;
      Foreach instruction $i$ $\in$ $cs$
      { compute $EST_i$ based on $I$'s dependences in the destination block
         $LFT_i$ = $LFT$ of the last instruction in the destination block }
         /* find overlapping resource holes */
      Foreach instruction $i$ $\in$ $cs$, in decreasing order of $EST_i$
      { Forall resources $r$ required by $i$
         { select holes $h_r$ such that they overlap with the other holes
              selected and with $i$
           if no such holes exist
           { undo all moves from the current critical set;
              return $reduce$; }
           Insert $i$ into $h$'s allocation chain; }
         Update the hole description information; }
      $reduce$ = $reduce$ + $\min_{i \in cs}$ ( $r_i$ ); }
   return $reduce$;
}
```

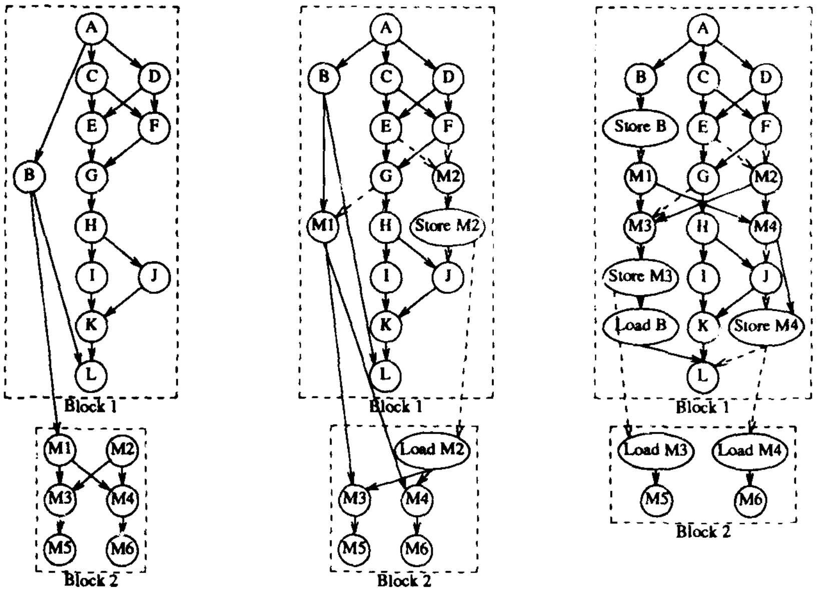
Figure 4: Function fill()

next critical set is {C, D}. Note that B is not in the critical set since moving it would not affect the length of the critical path.

If not all instructions in a critical set can be moved, none are moved. The allocation of resources is similar to that in local schedulers. Overlapping resource holes are found for all resources required by each instruction. However, the holes must be within the instruction's execution range, and wedged insertion is not performed, since the goal is to avoid increases to the critical path length of the destination block. An algorithm for performing global code motion in this manner is given in Figure 4.

Consider the problem of moving the instructions from Block 2 to Block 1 in Figure 5(a). Assume that there are three functional units and four registers available. The first critical set consists of instructions M1 and M2. Instruction M2 can be inserted in the functional unit hole following F and the register hole following D since the holes overlap. M2's value must be spilled since the register hole is not available to the end of the DAG. M1 can be inserted in the functional unit hole following B and the register hole following G, which results from killing F. The value computed by M1 need not be spilled. The resulting DAGs are shown in Figure 5(b). Next M3 and M4 are moved up. Since M4 is selected to kill both M1 and M2 it can use the same functional unit and register as M2. Instruction M3 can use M1's functional unit, and it will also take M1's register, forcing M1 to use B's register. B's value must now be spilled around the inserted instructions and M3's value is spilled before B's value is reloaded. The resulting DAGs are shown in Figure 5(c).

Traditional global schedulers, based on list scheduling, are able to identify functional unit holes. However, since the scheduler is separate from the register allocator, it does not know if there are registers available for the instructions that it moves up. Similarly, these schedulers cannot recognize when instructions from other blocks should be moved



(a) DAG of source instructions

(b) After moving two instructions

(c) After moving four instructions

Figure 5: Example of global code motion

up above instructions in the block with slack time, since these schedulers usually schedule all instructions in the current block first. Resource Spackling can move instructions from other blocks above instructions with slack time in the current block when overlapping resource holes are available.

Several problems concerning the insertion of fill instructions must be considered. The code motion algorithms must determine if the critical path length of the source block is decreased when loads of moved values are inserted in it. The algorithms must also consider the impact of code duplication on the critical paths.

6 Experimentation and Concluding Remarks

We have implemented Resource Spackling using our experimental compiler tool, pdgcc. Pdgcc is a C compiler front-end which performs dataflow and dependence analysis and generates intermediate code in the form of PDGs. Both local reductions and global code motions have been implemented. The target architecture of the experiments is a VLIW architecture that supports speculative execution. Two configurations of different numbers of functional units and registers are used. In the target architecture all instructions execute in one cycle, except for memory access instructions which execute in two cycles. Although the framework supports all code motions for conditionals, the experiments are based on code motions for speculative execution of instructions.

The procedures used consist of six routines from the C version of the linpack benchmark. Execution profile information was collected for each region. The resulting execution times, after local and global Resource Spackling has been performed, are determined by multiplying the execution times of each region by the number of cycles required to execute the region.

	VLIW machine 1		VLIW machine 2	
	4 FUs 8 Regs		8 FUs 16 Regs	
routine	local	global	local	global
matgen	2.48	2.64	2.48	2.73
dgcal	1.96	2.12	2.01	2.18
daxpy	2.24	2.35	4.09	4.40
ddot	2.12	2.12	2.12	2.12
idamax	3.01	3.01	3.01	3.01
epslon	2.17	2.38	2.17	2.38

Table 1: Experimental results

The results for two target architectures are shown in Table 1. The columns labeled `local` give the speedup over a 1 wide VLIW architecture when only local scheduling using reductions are performed. The columns labeled `global` give the speedups when both local and global scheduling are performed. The size of the critical path reductions ranged from 1 to 6 cycles and averaged 2.6. The number of instructions moved ranged from 1 to 8 and also averaged 2.6.

The numbers show that Resource Spackling is able to exploit the parallelism available in the benchmarks within the constraints of the architecture's resources. Further, in all but two cases, `ddot` and `idamax`, global code motion using Resource Spackling made additional improvements. Upon examining the routines and their profile information, it was discovered that instructions were moved during global code motion, but that neither their source or destination regions were executed during the profiling runs. The routines `ddot`, `idamax`, and `epslon` did not show any improvement when run on the 8 wide architecture because the resource requirement measurements showed that the 4 wide architecture provided sufficient resources.

Resource Spackling is a framework which is general enough to allow common architectural features to be incorporated. Separate $Reuse_R$ DAGs can be built for each type of resource provide, e.g., integer and floating point register files, and integer functional units and separate floating point adders and multipliers. Pipelines, resource demands that can be satisfied by several types of resources, and implicit resource demands also can be modeled in Resource Spackling[BGS94].

References

[AiN88] A. Aiken and A. Nicolau, A Development Environment for Horizontal Microcode, *IEEE Trans. on Software Engineering 14* , 5(May 1988) pp. 584-594.

[BeR91] D. Bernstein and M. Rodeh, Global Instruction Scheduling for Superscalar Machines, *Proc. Sigplan '91 Conf. on Programming Language Design and Implementation*, Toronto, Ontario, Canada, June 26-28, 1991, pp. 241-255.

[BGS93] D. A. Berson, R. Gupta and M. L. Soffa, URSA: A Unified ReSource Allocator for Registers and Functional Units in VLIW Architectures, *Proc. IFIP WG 10.3 Working Conference on Architectures and Compliation Techniques for Fine and Medium Grain Parallelism*, Orlando, Florida, Jan. 1993, pp. 243-254. (also available as University of Pittsburgh Computer Science Department Technical Report 92-21).

[BGS94] D. A. Berson, R. Gupta and M. L. Soffa, Representing Architecture Constraints in URSA, Technical Report 94-10, University of Pittsburgh, Computer Science Department, Feb. 1994.

[BEH91] D. G. Bradlee, S. J. Eggers and R. R. Henry, Integrating Register Allocation and Instruction Scheduling for RISCs, *Proc. Fourth International Conf. on ASPLOS*, Santa Clara, CA, April 8-11, 1991, pp. 122-131.

[Dil50] R. P. Dilworth, A Decomposition Theorem for Partially Ordered Sets, *Annuals of Mathematics 51* (1950) pp. 161-166.

[EbN89] K. Ebcioğlu and A. Nicolau, A *global* Resource-Constrained Parallelization Technique, *Proc. ACM SIGARCH ICS-89: International Conf. on Supercomputing*, Crete, Greece, June 2-9, 1989.

[Fis81] J. A. Fisher, Trace Scheduling: A Technique for Global Microcode Compaction, *IEEE Trans. on Computers C-30* , 7(July 1981) pp. 478-490.

[FoF65] L. R. Ford and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, N.J., 1965.

[GoH88] J. R. Goodman and W. Hsu, Code Scheduling and Register Allocation in Large Basic Blocks, *Proc. of the ACM Supercomputing Conference*, 1988, pp. 442-452.

[GuS90] R. Gupta and M. L. Soffa, Region Scheduling: An Approach for Detecting and Redistributing Parallelism, *IEEE Trans. on Software Engineering 16* , 4(April 1990) pp. 421-431.

[HsD86] P. Y. T. Hsu and E. S. Davidson, Highly Concurrent Scalar Processing, *Proc. 13th Annual International Symp. on Computer Architecture*, June 1986, pp. 386-395.

[MGS92] B. Malloy, R. Gupta and M. L. Soffa, A Shape Matching Approach for Scheduling Fine-Grained Parallelism, *Proc. 25th Annual International Symp. on Microarchitecture*, Portland, Oregon, Dec 1-4, 1992, pp. 264-267.

[MoE92] S. Moon and K. Ebcioğlu, An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors, Computer Science Research Report RC 17962 (#78691), IBM Thomas J. Watson Research Center, Yorktown Heights, NY, April 1992.

[NiG93] Q. Ning and G. R. Gao, A Novel Framework of Register Allocation for Software Pipelining, *Conf. Rec. 20th ACM Symp. on Prin. of Programming Languages*, Jan. 1993, pp. 29-42.

[Pin93] S. S. Pinter, Register Allocation with Instruction Scheduling: A New Approach, *Proc. Sigplan '93 Conf. on Programming Language Design and Implementation*, Albuquerque, NM., June 23-25, 1993, pp. 248-257.

[SHL92] M. D. Smith, M. Horowitz and M. Lam, Efficient Superscalar Performance Through Boosting, *Proc. 5th International Conf. on ASPLOS*, Boston, Massachusetts, Oct. 12-15, 1992, pp. 248-259.

[SwB90] P. Sweany and S. Beaty, Post-Compaction Register Assignment in a Retargetable Compiler, *Proc. of the 23rd Annual Workshop on Microprogramming and Microarchitecture*, Nov. 1990, pp. 107-116.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

An Approach to Combine Predicated/Speculative Execution for Programs with Unpredictable Branches

Mantripragada Srinivas[a] and Alexandru Nicolau[a] and Vicki H. Allan[b]

[a]Department of ICS, UC-Irvine. This work was supported in part by NSF grant CCR8704367 and ONR grant *N0001486K0215*.

[b]Department of Computer Science, Utah State University. This work was supported in part by NSF grant CDA-9300154.

Abstract: Predicated and speculative execution have been separately proposed in the past to increase the amount of instruction-level parallelism (ILP) present in a program. However, little work has been done to combine the merits of both. Excessive speculative execution can negatively affect the performance, especially when unguided by suitable profiler estimates. Similarly, predicated execution always results in unnecessary execution of operations present in the untaken branch. Conventional techniques use expensive profiler analysis to reduce the overhead costs of predicated execution. In this paper we *first study the effects* of individual as well as combined effects of speculative/predicated execution features. We then present an algorithm which attempts to combine the merits of the two. We show that for programs with unpredictable branch outcomes, the proposed technique improves the program performance as compared to previous techniques. We also show that separating the two (predication and speculation) results in less efficient schedules.
Keyword Codes: C.1.3, D.1.3, D.3.m
Keywords: instruction level parallelism, program dependence graph, region scheduling, predicated execution, speculative execution

1 Introduction

Architectures such as horizontal microengines, multiple RISC architectures, Superscalar and VLIW machines benefit from the utilization of fine-grain or instruction level parallelism (ILP). The presence of conditional branches in the code however limit the extent of ILP that can be extracted at a basic block level. Since many application programs are branch intensive and basic blocks do not contain many operations, the need to extend the scheduling phase beyond basic blocks is important in order to achieve better performance [4]. Techniques have been adopted by conventional compilers to parallelize the branch code. Two widely known techniques which attempt to alleviate this problem are *predicated* execution and *speculative* execution.

Predicated execution is presented as a technique to handle conditionals using basic block techniques. First, a branch condition is replaced with a statement which stores the result of the test in a predicate register, *P*. An instruction in the true branch such as

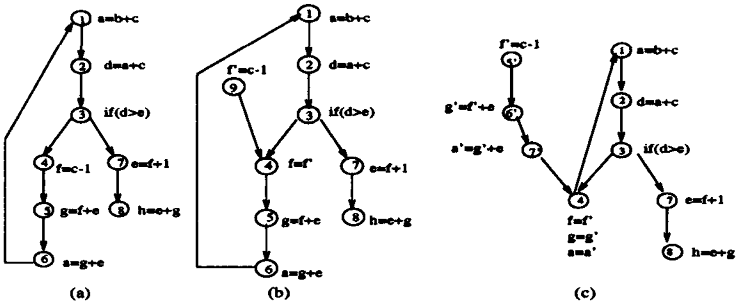


Figure 1: (a) Original data dependence graph.(b) After renaming $f = c-1$ (c) Dependence graph after renaming and forward substituting $g = f + e$ and $a = g + e$.

$a = b + c$ is replaced by a predicated instruction $a = b + c$ *if P_t* which specifies that the operation will actually be completed only if the predicate P is true. An instruction on the false branch such as *d=e*f* would be replaced by *d=e*f if P_f*. Mahlke et al. propose a parallel architecture which supports predicated execution in which the predicate operations may execute concurrently with the statement which assigns the predicate [6, 7]. This is possible because the operation is executed regardless of the value of the predicate, but is only allowed to change the assigned value if the predicate value is satisfied. Since the stores are performed in a later part of the execution cycle than the computation of the value, this is feasible. Thus, code containing branches is converted into straight-line, branch-free code. Warter et al. [5] later propose a reverse *if*-conversion process to convert the predicated representation back to the control flow graph representation in order to facilitate architectures without predicated execution support.

Speculative execution refers to the execution of operations *before* it is known that they will be useful. It is similar to predicated execution except that instead of allowing the operation to be performed at the same time step as the predicate, the operation can be performed many time steps before its usefulness is known. Renaming and forward substitution are used by Ebcioglu et al [8] and Nicolau et al [10] to move operations past predicates. The length of a dependence cycle containing a control dependence is reduced when the true dependencies in the cycle are collapsed by forward substitution. Renaming is a technique which replaces the original operation by two new operations, one of which is a copy operation while the other is the original operation but whose result is assigned to another variable. The copy operation copies the value from this new variable to the original variable. Since the new variable is used only in the copy operation, the assignment is free to move out of the predicate. Figure 1(b) shows how renaming is used to move the operation $f = c - 1$ past the predicate. A new variable f' is created and is assigned the result of the expression $c - 1$. In order to preserve the original semantics of the program, the value assigned to f' is copied back into f. Since f' is used only by the copy operation, it can move past the predicate. Figure 1(c) shows the resultant graph formed after renaming and forward substituting $g = f + e$ and $a = g + e$.

Various factors effect the desirability of using either predicated or speculative execution. If the code is highly constrained by data dependencies, predicated execution may be of little benefit. Speculative execution may perform better. In Figure 1(a), predicated execution would allow operations 3 and 4 to execute simultaneously which would have the effect of reducing the dependency cycle to five. However, Figure 1(c) shows that the cyclic dependency can be reduced to 4 by performing speculative execution only. Similarly, for branch intensive programs, predicated execution may perform better than speculative execution as it supports multi-way branching. Operations dependent on different predicates can be executed simultaneously, provided the parallel instruction can accommodate them. This provides more parallelism than speculative execution.

Both the techniques have their drawbacks as well. Excessive speculative or predicated operations may result in the generation of inefficient code. Speculated operations consuming extra resources may degrade the initial schedule. Similarly, predicated execution always examines operations from both the branches irrespective of which branch is finally taken. This may worsen the performance, if the predicted operations from the branch not taken consume extra cycles *after* the predicate has been computed.

The issue therefore becomes, how can the merits of both be combined in order to improve the efficiency of the generated code and if so, to what extent should they be combined? This paper attempts to answer these questions. In Figure 1(c), the cyclic dependency (4) after performing speculative execution can be further reduced to (3) with predicated execution (allowing 3 and 4 to execute concurrently).

The rest of the paper is organized as follows. Section 1.1 explains previous work done in this direction. Section 2 explains our model and the set of transformations we used for our comparison study. In this section, we also compare our combining algorithm with hyperblock scheduling [6], another relatively new technique which attempts to alleviate this problem. Finally, Section 3 shows the results obtained from the test cases.

1.1 Previous Work

An early attempt to speculatively execute predicated operations was done by the Impact Group [6]. They use hyperblocks to control the overhead which is inherent when *all* operations (which are performed conditionally) are predicated. Profiler estimates are used to analyze the various conditional paths and ignore blocks with low frequency. All conditionally executed operations within the hyperblock are initially predicated. The predicated operations are then considered for speculative execution. All the predicated operations which could not be speculated, either due to resource or dependence conflicts, remain as predicated operations. Global transformations are then performed on the resultant hyperblocks to generate parallel schedules. The performance of the technique depends on two main factors. First, the amount of regular parallelism a program contains i.e., how predictable and uniform are its control structures. Second, the overhead costs of compensation code (for the less frequently executed blocks) and other side-effects due to predicated exeuction, which may negatively affect the performance.

Another global scheduling technique which attempts to combine the merits of the two techniques is region scheduling proposed by Gupta et al. [11]. The extent of combining however is limited and less aggressive than hyperblock scheduling. A condition P_i is merged (predicated) with its true and false regions[1] R_j and R_k respectively, only, if both R_j and R_k contain sufficiently less parallelism to fully utilize the system resources. Precisely, the transformation is performed if $\delta_j + \delta_k \leq \delta_{ideal}$; δ_{ideal} is the number of function units available in the target architecture. The parallelism δ_i present in region R_i is measured as O_i/T_i, where O_i is the total number of operations present in the region and T_i is the execution time of the longest dependency chain.

[1] Regions represent independent schedulable units and summarize all statements having the same control condition. More details can be found in subsection 2.1.

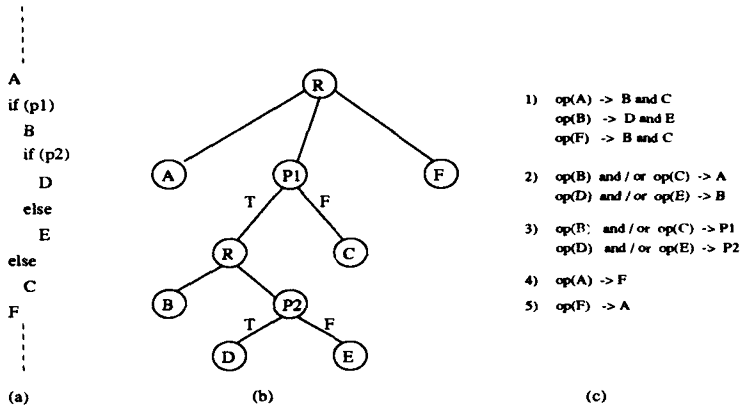
Most of the earlier work aimed at increasing the efficacy of using the speculative execution feature. Several hardware and software mechanisms have been proposed. Boosting, a hardware mechanism to support speculative execution, was proposed by Smith et al [2]. The work describes a hardware feature to support delayed and accurate reporting of exceptions caused by speculatively executed instructions. However, their work is based on a trace-driven simulation which does not support predicated execution features. Moreover, trace scheduling possesses serious limitations as shown in [11]. Bernestein et al [3] present an algorithm to selectively speculate instructions. Speculation is performed after non-speculated operations (if any) have been considered for possible code motion. The work assumes a machine with low issue rates (2-3). Their model does not support code duplication and provides limited speculative execution (one branch only). However, neither of the techniques support predicated execution.

2 Our Approach

We propose a technique which attempts to combine the merits of both the predicated and speculative execution features. Predicated execution is used to simultaneously execute control dependent operations with the computation of the predicate. Operations which can not execute simultaneously with the predicate are not predicated. Subsequent predication is unnecessary as the outcome of the branch condition will be known then. This avoids the consumption of extra resources needed to execute operations in the untaken branch. Let C be the set of all operations which are performed conditionally. Let C_s be the set of all operations which can be executed speculatively (if such execution will not increase the execution time of the non-taken branch). Let $C_o = C - C_s$ be the other operations which cannot be speculatively executed, either because of data dependence or because of resource conflicts. The operations of C_o can be executed (predicated) concurrently with the operation on which they are control dependent, can be skipped via branching, or can be executed (taken branch).

The technique treats the jump instruction as a predicated instruction and allows the possibility for its concurrent execution along with the predicate controlling its execution. We assume that the writback stage of program counter is guarded for predicated jump instructions. In case, the predicated jump instruction is nullified, the program counter is assumed to point to the next sequential address in memory. We perform this partial combining only when the cost of inserting this predicated jump instruction is still less than the cost to fully predicate all the conditional operations or to strictly disallow control dependent operations to execute simultaneously. The combining technique is performed only if both of the above two conditions is satisfied. More details about the technique are described in section 2.2.

We implemented the above technique in our enhanced version of the region model. The program dependence graph (PDG) [9] is used as a framework for performing our parallelizing transformations. The purpose of this study is two-fold. First, to show that predicated or speculative execution by itself cannot fully exploit the global ILP available in a program. Second, to show that the combining technique we propose, is better than previous techniques which attempted to combine the merits of both the predicated and speculative execution feature.



```
 :
 :
A
if (p1)
B
 if (p2)
  D
 else
  E
else
 C
F
 :
 :
```

```
1)  op(A) -> B and C
    op(B) -> D and E
    op(F) -> B and C

2)  op(B) and / or op(C) -> A
    op(D) and / or op(E) -> B

3)  op(B: and / or op(C) -> P1
    op(D) and / or op(E) -> P2

4)  op(A) -> F

5)  op(F) -> A
```

(a) (b) (c)

Figure 2: Code Motion Possibilities (a) Program Fragment (b) PDG Equivalent (c) Code Motion Possibilities

2.1 Our Transformations

We used our enhanced version of the region scheduler [1] as our model for performing all the parallelizing transformations. The transformations include *pipelining*, *loop* transformations and *code motion* transformations. Pipelining is used to overlap (simultaneously execute) operations present in different iterations of a loop body. Loop transformations peels one or more iterations from the loop body to a region just before/after the loop. This transformation is used to increase the parallelism present outside the loop. In our model, loop transformations are performed (if necessary) after pipelining to peel the iterations from the transformed loop body. Code motion transformations are then employed to redistribute the parallelism among regions. Figure 2(c) shows the various code motion transformations performed in our model. Figure 2(b) shows the PDG equivalent of a program fragment in Figure 2(a). Nodes A,···,F represent simple region nodes and contain operations. Nodes R represent combined region nodes as each has heterogeneous descendants. Nodes P represent the predicate condition operation. The various code motion transformations as listed in Figure 2(c) are given below. All the transformations are subjected to dependence constraints and resource availability.

1 **Code Duplication** Operations from region A can be moved to regions B and C. Similarly, operations from region B can be moved to regions D and E.

2 **Speculative Execution** Operations from region B and/or C can be speculatively executed at region A. Similarly, operations from region D and/or E can be speculatively executed at region B. An operation is speculatively executed only if both the renamed and the copy operation generated can be executed without increasing execution time.

3 **Predicated Execution** Operations from B and/or C can be executed simultaneously with the predicate condition operation P1. Similarly, operations from D and/or E can be executed simultaneously with P2.

4 **Delayed Execution** Operations from A can also be moved to F.

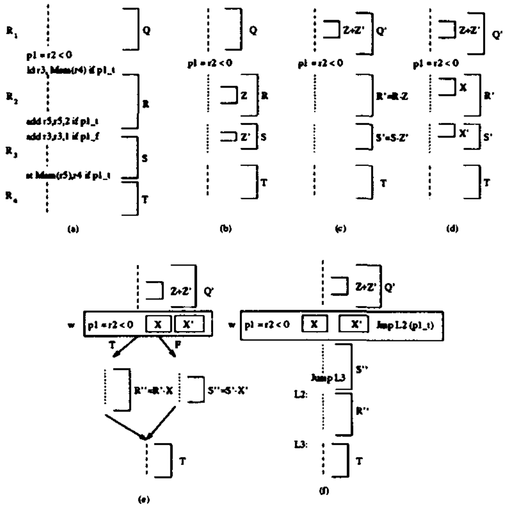


Figure 3: Combining Technique (a) Initial Code Fragment (b) Determination of predicated operations which can be speculated (c) After Speculative Execution (d) Determination of Predicated Operations that can be Accommodated in w (e) Combining Technique (f) Pseudo-code Equivalent.

5 Eager Execution Operations from F can be executed eagerly at A.

At each stage, operations with maximum height are considered first as possible candidates for code motion. Among operations with the same height, the one with more immediate breadth (successors) is selected next. We assume branch outcomes are unpredictable and therefore assume equal frequency of execution on both branches. The details and the style of each individual transformation can be found in [1]. Our previous model did not incorporate predicated execution capabilities. We implemented the proposed combining technique to facilitate predicated execution. We also implemented the combining technique proposed by the hyperblock method in our model and compared it with our technique.

2.2 Combining Algorithm

In this section, we explain our combining algorithm and then compare it with the similar work done at the hyperblock level.

Definition 1 A parallel instruction is said to be of width w if it can execute w operations in one instruction cycle. An instruction of width w can accommodate w regular (ALU/memory) operations along with w copy operations.

Definition 2 $P(X)$ represents the number of parallel instructions (of width w) required to execute *all* operations in set X.

Figure 3 illustrates the combining technique proposed by the hyperblock method in the framework of our region model. Figure 3(a) shows an example code fragment. Each region R_1, R_2, R_3 and R_4 contains Q, R, S, T operations respectively. Assume that both the branches (R_2, R_3) have equal likelihood of being executed and that the resultant hyperblock formed comprises of R_1, R_2, R_3, R_4. Initially, all operations R and S present in the true and false branch regions are predicated on condition $p1$. Assume, that only Z and Z' can be speculatively executed from R and S, owing to dependence and resource[2] constraints. Figure 3(c) shows the code after Z and Z' have been speculatively executed. In the hyperblock technique, all the remaining operations R' and S' which could not be speculated, are predicated. This may consume unnecessary resources especially when the code present in the untaken branch is sufficiently large. This can be avoided by branching to the appropriate region depending on the outcome of $p1$. We also make the instruction executing $p1$ as a predicated instruction in order to facilitate parallel execution of control dependent operations. Assuming that the width of the instruction is w, operations X and X' from the true and the false branch regions are executed in parallel along with p1, as shown in Figure 3(e). The equivalent pseudo-code formed is shown in Figure 3(f). A predicated jump operation is now inserted.

The entire sequence of steps is summarized as follows. The schedule cost estimates for each of the three cases (total predication, partial predication, no predication) are first determined. The resultant global schedules formed using the schedules at the region nodes (which are currently being used for analysis) are used as estimates for comparison purposes. The process is repeated for each of the above three cases separately. The one with a better estimate is chosen as the combining method and the process is repeated until all the regions are considered for scheduling. The specific combining equations used to achieve the above effect are given below.

$$P(R' + S' + \mathrm{p1}) > 1 + P(R'') \cdots true \tag{1}$$

$$P(R' + S' + \mathrm{p1}) > 1 + P(S'') \cdots false \tag{2}$$

$$0.5 * (P(R') + P(S')) > 1 + 0.5 * (P(R'') + P(S'')) \tag{3}$$

$P(R' + S' + \mathrm{p1})$ is the number of instructions required to execute operations R' (true), S' (false) and the condition operation (p1) as shown in Figure 3(c). $P(R'') + 1$ is the number of instructions required to execute the remaining operations (R") in the true branch plus the inserted predicated instruction as shown in Figure 3(f). Equations (1) and (2) are used to determine if the cost of performing partial predication/speculation as shown in Figure 3(f), is less than that of performing total predication, as shown in Figure 3(c). Equation (3) compares the costs of performing partial combining against following strict control dependence. The equations are multiplied by 0.5 to average the number of instruction cycles required for the true and false branch respectively. Partial combining is avoided if any of the above two conditions do not hold true. The average performance of the algorithm can be improved by using weighted equations and/or profiler estimates. We are currently studying the effect of the two in our combining heuristics. Our combining algorithm is shown in Figure 4(a).

[2] Resource constraints here refer to the availability of these operations (Z and Z') being accommodated for free in R_1.

```
Global Transfromation Phase
{
    perform loop transformations          ··················· Phase - I
    for every region/predicate node in the
    PDG traversed in inorder traversal    ·············· Phase - II
    {
     If predicate node
     {
         consider code motion possibilities  ·········· Figure 2
         If equations (3) is satisfied
             If equations (1) and (2) are satisfied
                 perform partial predication ·········· Figure 3(f)
             else
                 perform total predication ············ Figure 3(c)
         else
           perform neither partial/total predication
     }
     If region node
         consider code motion possibilties  ·········· Figure 2
    }
}
```

(a)

Issue rate = 3

Test Cases	PE	SE	COMBINED	
			ERS1	ERS2
	%S	%S	%S	%S
Mandelbrot	42.3	54.6	52.1	66.8
Bresnham	35.1	55.4	56.7	56.7
Minmax	30.2	20.5	30.2	51.8
Tomcatv	40.7	53.9	52.6	68.4
Pterm_ops	37.3	37.3	37.3	56.4
Kernel 16	28.6	44.3	36.5	44.3
Kernel 17	41.9	65.3	42.7	65.3
Kernel 20	41.2	55.7	47.2	55.7
Average	33.5	48.4	44.5	58.2

(b)

Figure 4: (a) Combining algorithm (b) Results Obtained for an Issue Rate = 3.

3 Performance Evaluation

3.1 Evaluation Methodology

The underlying architectural model is a VLIW processor with homogeneous function units. The latency of the ALU, memory (load/store) and branch operations is assumed to be one, two and one respectively. Our current enhanced region scheduling (ERS) compiler version does not handle procedure calls and pointer references. We also assume that all the references to the cache are perfect. Thus, our results compare the relative CPU execution times. An instruction of width w (initially assumed) can accommodate w regular (ALU, memory, branch) operations along with w copy operations. All the results are derived using a global list-based scheduler which schedules each region/predicate node in an inorder fashion. The performance is measured as the number of instruction cycles required to schedule the whole program. The speedup is measured as the ratio of the number of cycles required for sequential execution over the number of cycles required after performing parallelizing transformations.

The benchmarks consist of programs from the SPEC benchmarks, livermore loops and other commonly used application programs. All the programs are written in C and contain loops with one or more conditionals. In order to present a fair comparison with previous methods, we avoided loop pipelining transformation in our analysis. However, our combining algorithm can be used to treat pipelined loops as well. Peeled iterations from the transformed (pipelined) loop body containing conditionals can be treated with non-loop conditional constructs. In our analysis, loop peeling is performed first (if necessary) to peel the iterations from the loop body. This transformation provides more opportunities for code motion with the code present outside the loop body. Code motion transformations are then performed to redistribute the parallelism among regions. The proposed combining technique is applied during the code motion stage.

Issue rate = 5

Test Cases	PE	SE	COMBINED	
			ERS1	ERS2
	%S	%S	%S	%S
Mandelbrot	55.8	55.8	57	68.1
Bresnham	55.4	55.4	59.4	59.4
Minmax	30.1	32.5	31.3	62.6
Tomcatv	43.4	53.9	59.2	71.1
Pterm_ops	45.6	50.1	53.6	60.1
Kernel 16	58.8	67.8	65.3	70.1
Kernel 17	58.9	67.9	65.3	67.9
Kernel 20	56.6	64.3	47.3	64.3
Average	50.1	56	54.8	65.5

Issue rate = 7

Test Cases	PE	SE	COMBINED	
			ERS1	ERS2
	%S	%S	%S	%S
Mandelbrot	66.8	68.1	66.8	69.3
Bresnham	56.7	58.1	59.4	59.4
Minmax	31.3	53.1	31.3	62.6
Tomcatv	44.7	55.3	59.2	71.1
Pterm_ops	46.3	54.5	60.1	71.8
Kernel 16	65.2	72.3	69.4	82.7
Kernel 17	66.9	67.9	66.9	67.9
Kernel 20	56.9	64.3	47.3	64.3
Average	54.4	61.7	57.8	68.7

Figure 5: (a) Results Obtained for an Issue Rate = 5 (b) Results Obtained for an Issue Rate = 7.

3.2 Results

Consider Figure 4(b) and Figure 5 which show the performance increase for issue rates of 3, 5 and 7 respectively. The issue rate is the maximum number of operations that can reside in a single parallel instruction. PE shows the performance increase over sequential execution using predicated execution support. All other transformations except speculation are performed for the case study. SE shows performance increase using speculative execution support. Predicated execution support is avoided for this case study. Finally, ERS1 (hyperblock) and ERS2 (proposed), *the two different combining* approaches are also shown. It should be noted that the amount of global speculative code remains the same for SE, ERS1 and ERS2. The performance difference for the three models comes in, how the remaining unspeculated operations in each branch region are treated. SE disallows control dependent operations to execute simultaneously. ERS1 allows control dependent operations to execute simultaneously but predicates all the remaining unspeculated operations present in the branch regions. ERS2 performs partial combining as discussed in Section 2.2.

The results show that SE performs better than PE for most of the test cases except for the test case minmax. For the test case (Minmax) with multiple nested predicates (and few conditionally executed operations), predicated execution has a greater performance than speculative execution for lesser issue rates. Less speculative execution is performed for low issue rates due to insufficient resources. However, for higher issue rates, the performance is greater for speculative execution. On the average, speculative execution performed 5%-16% better than using predication execution only.

It is interesting to note that SE performed better than ERS1 for most of the test cases. This clearly shows that performing total predication of the remaining unspeculated operations may result in the consumption of extra cycles. The effect was more pronounced for low issue rates and less for higher issue rates. An average of 2-3 peeled iterations from the loop body was observed for most of the test cases. For test cases tomcatv and pterm_ops, ERS1 performed consistently better than SE. This shows that for branch intensive programs with less side-effects (due to predicated execution), ERS1 can actually find more parallelism than SE. However, the side effects seem to be more pronounced for other test cases, thereby affecting the performance of ERS1. ERS2 seem to perform consistently better than all the other models. On the average, the performance improvement of ERS2 is around 10%-15%, 5%-20%, 15%-25% over using ERS1, SE and PE respectively. For some of the test cases (kernel 20), the performance of ERS2 was similar to SE. The proposed technique was never performed for that test case resulting in a similar performance. Similarly, for the test case (bresnham), ERS2 and ERS1 had a similar performance.

4 Summary and Conclusions

In this paper, we have studied the effects of individual as well as combined effects of predicated and speculative execution features. Previous studies have separately proposed several hardware/software mechanisms to improve their existing performance. We show that both predicated and speculative execution are important features and should be employed by any parallelizing compiler in order to extract all the potential parallelism available in a program. We also presented a technique which attempts to combine the merits of the both and showed that the proposed technique performs better than the technique proposed by the hyperblock method. All our transformations are performed using the extended region model [1].

References

[1] V. H. Allan and J. Janardhan and R.M. Lee and M. Srinivas "Enhanced Region Scheduling on a Program Dependence Graph," *Proceedings of the 25th International Symposium on Microarchitecture (MICRO-25)*, pp. 72-80, December 1-4, 1992.

[2] M. D. Smith and M. A. Horowitz and M. S. Lam "Efficient superscalar performance through boosting," *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp 248-259, October, 1992.

[3] D. Bernstein and M. Rodeh "Global Instruction Scheduling for Superscalar Machines," *Conference Record of SIGPLAN Programing Language and Design Implementation*, pp 241-255, 1991.

[4] A. Nicolau and J. A. Fisher "Measuring the available parallelism for very long instruction word architectures," *IEEE-TC*, pp 1088-1098, September, 1984.

[5] N. J. Warter and S. A. Mahlke and W. W. Hwu and B. R. Rau "Reverse If-Conversion," *Conference Record of SIGPLAN Programing Language and Design Implementation*, June, 1993.

[6] S.A. Mahlke and D.C. Lin and W.Y. Chen and R.E. Hank and R.A. Bringmann "Effective Compiler Support for Predicated Execution Using the Hyperblock," *Proceedings of the 25th International Symposium on Microarchitecture (MICRO-25)*, pp 45-54, December 1-4, 1992.

[7] B. R. Rau and D. W. L. Yen and W. Yen and R. A. Towle "The Cydra 5 Departmental Supercomputer: Design Philosophies," The IEEE *Computer*, pp 12-25, January 1989.

[8] K. Ebcioğlu and T. Nakatani "A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture," *Languages and Compilers for Parallel Computing*, pp 213-229, MIT Press, Cambridge, MA, 1990.

[9] J. Ferrante and K.J. Ottenstein and J.D. Warren "The Program Dependence Graph and Its Use in Optmization," *ACM Transactions on Programming Languages and Systems*, Vol. 9, pp 319-349, July, 1987.

[10] A. Nicolau and R. Potasman and H. Wang "Register Allocation, Renaming and Their Impact on Fine-Grain Parallelism," *Proceedings of the 4'th International Workshop on Languages and Compilers for Parallel Processing*, August, 1991.

[11] R. Gupta and M.L. Soffa "Region Scheduling: An Approach for Detecting and Redistributing Parallelism," *IEEE Transactions on Software Engineering*, Vol. 16, pp 421-431, April 1990.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

A PDG-Based Tool and Its Use in Analyzing Program Control Dependences*

Chris J. Newburn, Derek B. Noonburg, and John P. Shen; {newburn, derekn, shen}@ece.cmu.edu

Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213

Abstract: This paper explores the potential of a program representation called the program dependence graph (PDG) for representing and exposing programs' hierarchical control dependence structure. It presents several extensions to current PDG designs, including a node labeling scheme that simplifies and generalizes PDG traversal. A post-pass PDG-based tool called PEDIGREE has been implemented. It is used to generate and analyze the PDGs for several benchmarks, including the SPEC92 suite. In particular, initial results characterize the control dependence structure of these programs to provide insight into the scheduling benefits of employing speculative execution, and exploiting control equivalence information. Some of the benefits of using the PDG instead of the CFG are demonstrated. Our ultimate aim is to use this tool for exploiting multi-grained parallelism.

1. Introduction

A program representation called the program dependence graph (PDG) [6] explicitly represents all control dependences, and readily identifies control-equivalent basic blocks that may be far away from each other in the CFG. The hierarchical representation of the program control dependence structure in a PDG allows a large fragment of the code to be treated as a single unit. Since the work by Ferrante et al. [6], a number of other researchers have studied the use of PDG for code motion [8, 2, 5], program partitioning [13], code vectorization [4], register allocation [11], program slicing and software engineering [9,12], and code translation for dataflow machines [3].

PDGs appear well-suited for the exploitation of program parallelism of different granularities. Current compilers exploit fine-grained parallelism by overlapping the execution of instructions within a basic block or a small number of adjacent basic blocks, and medium-grained parallelism by overlapping the execution of loop iterations. Recent studies [10] suggest that additional parallelism can be harvested if the compiler is able to exploit control equivalence and use multiple instruction streams. This parallelism may not be strictly fine-grained or medium-grained. Multi-grained parallelism is an informal generalization of parallelism that can be achieved by the parallel execution of code fragments of varying sizes or granularities, ranging from individual instructions to loop iterations, to mixtures of loops and conditionals, to large fragments of code. PDGs appear to be well-suited for exploiting such parallelism, since such fragments are subgraphs of the PDG that can be hierarchically summarized, and because they explicitly maintain accurate control dependence information.

Our eventual goal is to develop a PDG-based compilation tool capable of performing powerful code transformations to exploit multi-grained parallelism. This paper presents our initial results, most of which focus on programs' control dependence structure. It proposes a number of extensions to the current PDG design in order to facilitate more generalized code motion of varying granularities, for even unstructured and irreducible programs. A node labeling scheme is introduced that simplifies and generalizes PDG traversal.

The initial portion of a prototype PDG-based tool called PEDIGREE is complete. It is a post-pass tool which can be used in conjunction with existing compiler front ends or disassemblers to perform optimizations on existing source or object code. Currently, PEDIGREE is able to parse large, unstructured and irreducible programs into its internal PDG representation. The PDGs of these programs are analyzed, and their control dependence structures are characterized. Section 2 introduces the PDG and presents our extensions to the PDG. The PDG node labeling

* Supported by National Science Foundation Graduate Research Fellowships, and ONR contract No. N0001491-J-1518.

scheme is summarized in Section 3. Section 4 briefly describes the implementation of PEDIGREE. Section 5 analyzes the PDGs generated by PEDIGREE for a number of benchmarks, including some from the SPEC92 suite. The advantages of using the PDG over the CFG are highlighted using collected statistics in Section 6. Section 7 provides a summary and suggests potential code transformations that can be implemented based on PEDIGREE.

2. The Program Dependence Graph

The program dependence graph [6] is essentially a hierarchical control dependence graph (CDG) and data dependence graph. The explicit representation of control dependences facilitates the identification of control-equivalent code. The hierarchical control dependence structure identifi and groups code constructs for special processing.

2.1. PDG Definitions

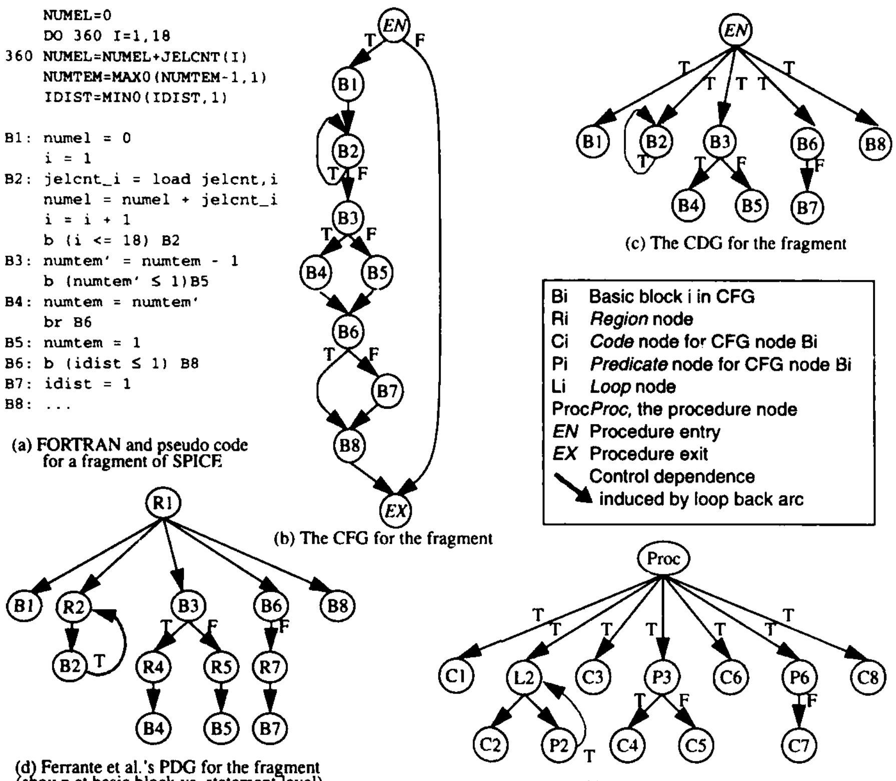
A control flow graph (CFG) represents a single procedure. Nodes in a CFG are basic blocks. An arc between nodes X and Y indicates one of the following: (a) a conditional or multi-way branch from X to Y, (b) an unconditional branch from X to Y, (c) X does not end with a branch and is immediately followed by Y, (d) X ends with a procedure call, and Y is the return target of the call. Only type (a) arcs represent true control dependences. They are traversed only on a specified condition, while the traversal of arcs of type (b), (c), and (d) does not depend on any condition. Figure 1b shows a typical CFG for the code in Figure 1a.

Nodes in a control dependence graph (CDG) may also be basic blocks. In a CDG, however, an arc from node X to node Y indicates that Y is immediately control dependent on X. This means that X ends with a conditional or multi-way branch, and that Y's execution depends on the value that determines the path out of X. This value is indicated by the arc label. Figure 1c shows the CDG derived from the CFG in Figure 1b. In the CFG, B6 follows B3, B4, and B5. However B6 is not control dependent on any of these; in fact, it is control equivalent to B3. CDG siblings that have exactly the same incoming arcs are control equivalent.

The program dependence graph (PDG) extends the CDG with hierarchical summary and data dependence information. The PDG has additional nodes and arcs to represent hierarchical relationships. PDG nodes hierarchically summarize regions of the CDG, rather than summarizing strongly-connected CFG components like a hierarchical task graph [7]. PDG nodes are classified into three basic types [8]. *Code* nodes represent basic blocks, and are leaf nodes in the PDG. The control dependences in the CDG are represented in arcs which emanate from *Predicate* nodes. *Region* nodes group together all nodes which share a common set of control dependences. Each *Code* node contains the set of instructions from the corresponding basic block. Conditional branch instructions are associated with *Predicate* nodes rather than *Code* nodes, since descendants of *Predicate* nodes are control dependent on the branch. Instructions can be broken into smaller sets, with *Code* nodes representing sub-blocks. Other PDG implementations, e.g. [2,4,6,8,13], generally use leaf nodes that represent statements rather than basic blocks. The representation of data dependences is orthogonal to the control dependence structure. Data dependences may be represented only within basic blocks if control flow order is maintained, or the representation may be as sophisticated as Gated Single Assignment form [3].

PDG arcs represent two distinct concepts. *Control dependence arcs* represent control dependences, and thus must emanate from a *Predicate* node. They correspond to the arcs of the CDG, and are labeled similarly. Conditional branches use two arc labels, T and F. Multi-way branches, e.g. from case statements, have a unique arc label for each branch target. Unlabeled arcs, which emanate from *Region* nodes, are called *grouping arcs*. They represent transitive, rather than immediate, control dependence for a set of nodes with a common set of control dependences. Arcs which emanate from the same node can be ordered according to the original control flow order [8]. This ordering is necessary only to preserve inter-node data dependences if they are not explicitly represented. It also simplifies data flow analysis and allows regeneration of a CFG identical to that from which the PDG is generated.

Figure 1d shows the original PDG representation [6] constructed from the CDG in Figure 1c. It has the same overall structure, but *Region* nodes have been added for each set of CDG nodes with control dependences in common.



```
    NUMEL=0
    DO 360 I=1,18
360 NUMEL=NUMEL+JELCNT(I)
    NUMTEM=MAX0(NUMTEM-1,1)
    IDIST=MINO(IDIST,1)

B1: numel = 0
    i = 1
B2: jelcnt_i = load jelcnt,i
    numel = numel + jelcnt_i
    i = i + 1
    b (i <= 18) B2
B3: numtem' = numtem - 1
    b (numtem' ≤ 1)B5
B4: numtem = numtem'
    br B6
B5: numtem = 1
B6: b (idist ≤ 1) B8
B7: idist = 1
B8: ...
```

(a) FORTRAN and pseudo code for a fragment of SPICE

(b) The CFG for the fragment

(c) The CDG for the fragment

(d) Ferrante et al.'s PDG for the fragment (shown at basic block vs. statement level)

(e) PEDIGREE's PDG for the fragment

Figure 1. (a) FORTRAN example from spice, and its (b) CFG, (c) CDG, (d) PDG as specified in [6], and (e) our PDG

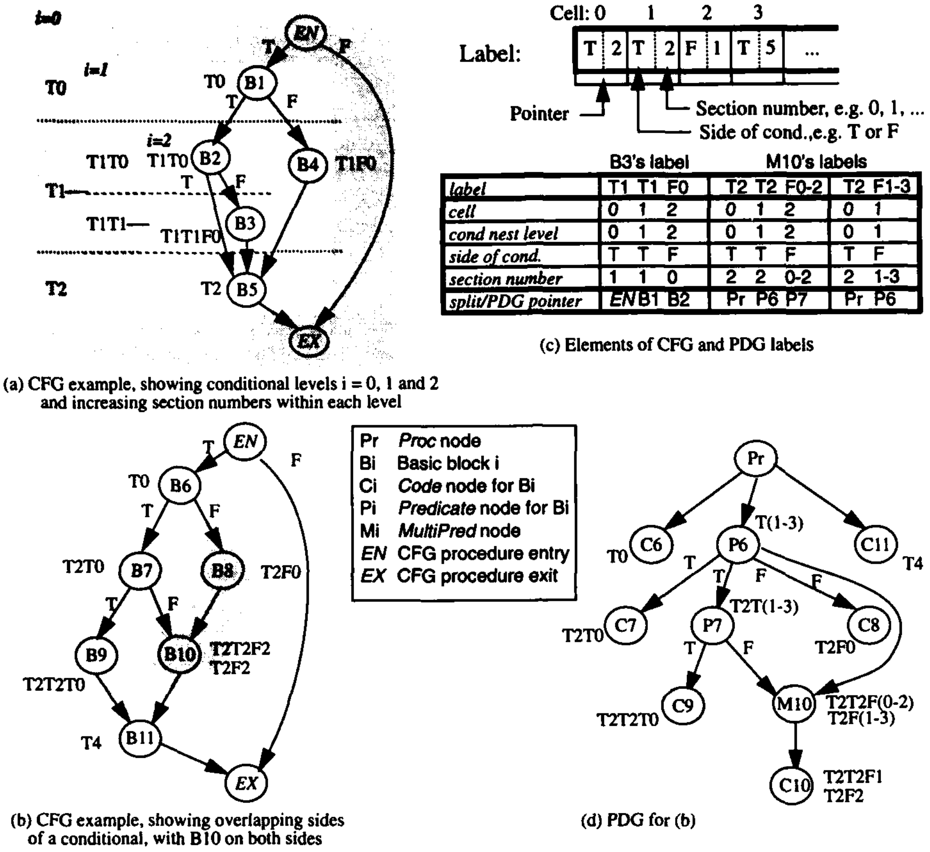
2.2. PDG Extensions

This work extends the PDG to better support exploitation of multi-grained parallelism. First, the *Region* node type described above is subclassified. This is done to identify and isolate PDG phenomena like cycles, regions with multiple control dependences, and irreducible loops, so that they can be handled more efficiently. Second, new node types are added to represent procedure calls and procedure entry points. Finally, each PDG node is labeled to indicate its position within the PDG. These labels facilitate efficient PDG traversal and computation of inter-node relationships such as dominance, reachability and control dependence. The first two extensions are presented in the following subsections; node labeling is presented in Section 3. Figure 1e exemplifies this type of PDG.

2.3. Subclassification of Region Nodes

The generalized *Region* node type is subclassified to represent certain control dependence structures which should be handled differently by the compiler. The special treatment of the region associated with one of these special *Region* nodes is indicated by the region type.

Multiple-predecessor (*MultiPred*) nodes group sets of nodes that share multiple immediate control dependences. *MultiPred* nodes can arise from unstructured control flow like goto and break statements. A *MultiPred* node signals the compiler that special actions, such as code replication or liveness analysis, must be taken when moving



(a) CFG example, showing conditional levels i = 0, 1 and 2 and increasing section numbers within each level

	B3's label	M10's labels	
label	T1 T1 F0	T2 T2 F0-2	T2 F1-3
cell	0 1 2	0 1 2	0 1
cond nest level	0 1 2	0 1 2	0 1
side of cond.	T T F	T T F	T F
section number	1 1 0	2 2 0-2	2 1-3
split/PDG pointer	*EN* B1 B2	Pr P6 P7	Pr P6

(c) Elements of CFG and PDG labels

(b) CFG example, showing overlapping sides of a conditional, with B10 on both sides

(d) PDG for (b)

Figure 3. Labeling examples. (a) CFG for structured code, (b) CFG for unstructured code, (c) Elements of CFG and PDG labels, (d) PDG for the CFG in (b).

Figure 3a illustrates conditional nesting levels. Split *EN* is at level 0, B1 is at level 1, and B2 is at level 2. Figure 3c illustrates the makeup of the label for B3, T1T1F0. It has three cells since it has three enclosing conditionals. Cell 0, T1, describes B3's position within the outermost conditional: it is on the true side and it is in section 1. B1 is in section 0 and B5 is in section 2 at nesting level 0. Cell 1, T1, describes B3's position within the enclosing conditional whose split is at level 1, B1. It is on the true side, and in section 1. Because B3 shares a common prefix, T1—, with B2, and it has a larger section number than B2 in cell 1, it must be reachable from B2. Finally, it is the first node in the innermost conditional nesting level, and thus has a section number of 0 in the last cell. Cell 1 of B4's label, F0, indicates that it is on the false side of the enclosing conditional at level 1.

Standard programming constructs such as if-then-else, multi-way branches and loops form structured code. Unstructured code allows control to flow from one side of a conditional to another, such that some nodes in that conditional are on more than one side of the conditional. A label can only describe position with respect to one side of a conditional, so such nodes are assigned one label for each side of a conditional that they are on. In Figure 3b, a conditional is formed by the split node B6 and structured join B11, which are at level 1. The side of this conditional associated with the T arc from B6 has nodes B7, B9 and B10. The other side of the conditional consists of

If a PDG is constructed for every procedure in a program, then a *Call* node can contain a pointer to the called procedure's PDG. An implicit call graph of the program is then present, with the PDGs as nodes, and the procedure pointers within *Call* nodes as arcs. Summary information for a procedure can be maintained in the *Proc* or *Call* nodes to support inter-procedural analysis and demand-driven inlining.

With this PDG design, a code fragment of varying size and granularity, including a mixture of control-equivalent regions of different granularities, can be summarized by a *Predicate* or *Region* node and treated as a single unit. This enables these fragments to be moved past other such fragments in what can be called multi-grained code motion. Identification of different types of regions indicates how each region should be treated. The next section describes a node labeling scheme that facilitates generalized PDG traversal for [illegible] code motion.

3. Node Labeling and PDG Construction

A node labeling scheme is proposed which labels each node in a PDG, as well as in a CFG, so that its position relative to any other node may be determined by comparing their labels. This facilitates computation of reachability, dominance, and control dependence relationships, as well as node-to-node traversals. Without such a labeling scheme, these operations would require expensive traversals from the root to each node. The labeling scheme presented here handles unstructured and irreducible code. In addition, it allows inserted nodes to be labeled without global changes to other node labels. Finally, a labeled PDG can be constructed during CFG labeling with minimal additional computation. A brief description of the labeling scheme follows.

3.1. Program Control Dependence Structure

The structure of a program is determined by its patterns of control flow. There are three important control flow patterns: splits, joins, and back arcs. A *split* is a CFG node with multiple successors because of a conditional or multiway branch. It has one successor for each branch target, or "side", of the conditional. For example, an if-then-else construct has a split with two successors, one for the "then side" and one for the "else side", and a case statement has a split with one side for each case. *Joins* are CFG nodes with multiple predecessors. These occur when two or more paths rejoin. Each split has a corresponding *structured join*, which is the first CFG node where all paths from that split rejoin. In structured code, all joins are structured joins. In Figure 3b, the splits are *EN*, B6, and B7. The structured joins for these splits are *EX* for *EN*, and B11 for both B6 and B7. B10 is not a structured join. A back arc is a control-flow arc such that the tail of the arc has a higher depth-first number [1] than the head.

Nodes must be labeled such that a comparison of the labels indicates reachability, dominance, and control dependence. Node Y is *reachable* from node X iff they are on the same side of all enclosing conditionals and Y is on some path from X to *EX* that does not include back arcs. X *dominates* Y iff all paths to Y from *EN* pass through X. Y is *control dependent* on X if X is a split, and Y is inside that split's conditional. In order to evaluate these relations, the following information must be maintained at each node: 1) the conditionals it is nested within, 2) which sides of each enclosing conditional the node is on, and 3) relative position on each of those sides. Conditionals may be nested, producing the different conditional nesting levels illustrated in Figure 3a. The *EN* and *EX* nodes are at level 0. A node at conditional nesting level *i* is nested within *i* conditionals.

3.2. Node Labeling

A label consists of an array of cells, where cell *i* describes the node's position within the enclosing conditional whose split is at nesting level *i*. Each cell has two fields. The first specifies which side of that conditional the node is on, e.g. with a truth value of T or F. The second is the section number, which indicates the relative position on that side of the conditional. It can take on values like 0,1,2... . When inserting a new node between two existing nodes, non-integer values can be used for section numbers to avoid having to modify section numbers of any existing nodes. Section numbers increase in the forward direction. Each cell also has a pointer to the CFG split or immediate PDG ancestor in the innermost enclosing conditional. This pointer aids in node labeling and PDG traversal of unstructured code. Figure 3c shows the elements of a label, and shows the correspondence of labels' cells to nodes with three examples.

nodes B8 and B10. B10 is on both sides of the conditional because arcs B7-B10 and B8-B10 bridge between sides of the conditional. Thus B10 has two labels, one for each side of B6, with opposite truth values in cell 1.

Labels of ***Predicate*** and ***Region*** nodes have ranges for their section numbers in the last cell, *l*, instead of single values. The range bounds the possible section number values for cell *l* in the labels of the descendants of that node. Hence the labels of the root of a region can be inspected during traversal to determine if a particular labeled node is in that region. For example, in Figure 3d, all of P7's descendants have a label in the range (T2T1, T2T3), abbreviated T2T(1-3). The section number range is distinct from the multiple labels associated with *MultiPred* nodes. For example, M10 has two labels, T2T2F(0-2) and T2F(1-3).

3.3. PDG Construction

CFG node labels are assigned in depth-first order [1]: each node is labeled prior to its successors. The label(s) for a node are constructed from its predecessors' labels. Labels are essentially a list of conditionals whose splits have not been rejoined. Labels have a new cell appended for every conditional their node is nested within. At a structured join, labels from predecessors on all sides of a conditional are merged into a single, shorter label.

A PDG and its node labels may be constructed as the CFG is labeled. After a CFG node is labeled, a PDG *Code* node is created, along with a *Predicate* or *Call* node if the CFG node ends with a conditional branch or call instruction, respectively. A *Loop* or *IrrLoop* node is generated for each loop entry, and *MultiPred* nodes are generated for nodes with multiple control dependences. *Code* nodes are assigned the labels of the corresponding CFG nodes.

The immediate control dependence of CFG node X_C on the split node S_C of its innermost enclosing conditional is indicated by the last cell of X_C's label. A control dependence arc is added in the PDG from the *Predicate* node S_P that corresponds to S_C to the PDG node X_P that corresponds to X_C. The last cell of X_P's label points to S_P. In Figure 3, B6 has corresponding *Code* and *Predicate* nodes C6 and P6. C6 has the same label as B6, while P6's labels bound the labels of *C7-C10*. Since C10 has multiple immediate control dependences, a *MultiPred* node M10 is created for it. As seen in Figure 3c, both of M10's labels have their pointers set to P6 in cell 1. The section number ranges for *Predicate* and *Region* node labels are assigned after all of their children have been constructed.

3.4. Uses of Node Labels

With this node labeling scheme, evaluation of reachability, dominance, and control dependence are quite efficient. Let A[*i*] be the i^{th} cell of label A, and let *d* be the first cell index for which labels A and B differ. A < B if A[*d*] and B[*d*] have the same truth value, and A[*d*] has a lower section number. For example, T2T1 < T2T2F1 but T2T0 $\not<$ T2F2. Node B is reachable from A iff there exists a label of A, A_i, and a label of B, B_j, such that $A_i < B_j$. In Figure 3a, B2 is reachable from B1 since T0 < T1T0. B2 (T1T0) is not reachable from B4 (T1F0) because their labels have different truth values in the first differing cell. Node A dominates node B iff for all of B's labels, B_j, there exists a label of A, A_i, such that $A_i < B_j$ and the first cell in which A_i and B_j differ is the last cell of A_i. The definition of postdominance is similar. In Figure 3b, B7 does not dominate B8 because they are on different sides of a conditional. B6 dominates B10, but B7 does not dominate B10. B10 postdominates B8. Control dependence is indicated with a pointer in cell *i* to the split at conditional nesting level *i*, on which the node is control dependent.

Labels uniquely describe a location in the PDG that is a target for traversal, since no two nodes have the same labels. PDG traversal from node *N* toward a target label *T* proceeds one PDG node at a time. The direction from *N* — right, left, up or down — is determined by comparing *T*'s label(s) with the label(s) of *N* and its neighbors. If the next neighbor in the direction of traversal is past the target, traversal stops. Unstructured code creates multiple paths in the PDG, between a *MultiPred* and the first *Predicate* that dominates the code in the *MultiPred* region. Non-overlapping traversal along multiple paths is very difficult without guidance from a labeling scheme. Consider traversal from M10 in Figure 3d to C6 via P6, along the two paths. P6 is the first node common to all paths up from M10, as determined by the pointers to P6 in M10's labels, shown in Figure 3c. Non-overlapping traversal from M10 along one path proceeds only up to P6 without visiting it, while the second should then proceed through P6 to C6. The target for traversal along the first path should be inside P6's region but before all of its children, such as T1.5, since T1 < T1.5 < T2—. The target label for the other path toward C6 remains T0. T0 is before T1, so traversal continues to the left of P6. Such traversal is important for code motion, since it allows multiple copies of code moved out of a *MultiPred* region to be unified at the dominating *Predicate*. Section 5 addresses the complexity of label comparison and PDG construction. It uses collected data to illustrate their efficiency.

4. PEDIGREE: A PDG-Based Framework

The implementation of a post-pass, PDG-based, retargetable compilation environment called PEDIGREE is in progress. PEDIGREE is implemented in C++, following an object-oriented design philosophy. The unique treatment of each kind of PDG node, for summary, traversal, and scheduling, is encoded into the members and member functions of different classes. The software architecture has four layers. Lower layers generally provide services to higher layers. At the bottom, the *basic layer* handles fundamental data structures such as graphs. The *representation layer* deals with the CFG and PDG program representations. This includes constructing the CFG and PDG, and providing generic traversal functions. The *scheduling layer* provides low-level transformations related to code motion, including functions to perform data dependence analysis and memory disambiguation, as well as functions to actually move regions in the PDG. At the top, the *guidance layer* makes decisions regarding scheduling heuristics. This includes identifying parallelism-lean regions and attempting to fill them with code from parallelism-rich regions. The first two layers have been completed. Implementation of the third is nearing completion.

Currently, PEDIGREE performs the following functions. It parses assembly files generated by a high-level language compiler for an instruction set architecture specified in the architecture description file. It constructs a CFG for each procedure, constructs a labeled PDG, analyzes the PDG, and generates statistics. It also regenerates the CFG from the PDG in order to verify correct PDG construction. PEDIGREE has been used to collect the data for the SPEC92 benchmarks. It is currently targeted to the DEC Alpha 21064, but can be easily retargeted.

5. PDG Analysis

The current implementation of PEDIGREE has been used to analyze several benchmarks. A subset of ten benchmarks, including nine from the SPEC92 suite, are selected for presentation here. This section presents initial results for these benchmarks in two areas: general PDG statistics and PDG construction and labeling complexity.

5.1. General PDG Statistics

Table 1 lists the benchmarks used in this paper, and presents the distribution of the different types of PDG nodes. The presence of *MultiPred* nodes indicates that compilers must be able to handle the unstructured code in these benchmarks. The number of *MultiPred* nodes may be unexpectedly high for some of these benchmarks, but un-

Table 1: PDG Statistics

source	Bench-mark	lines of assembly	# pro-cedures	# PDG nodes	# PDG arcs	# *Code* nodes	# *Pred-icates*	# *Call* nodes	# *Multi-Preds*	# *Loop* nodes	# *IrrLoops*	Multi-way br, (avg # targets)
SPECfp92	ear	16902	113	2812	2849	1511	504	578	38	106	0	2 (18)
	nasa7	10095	18	985	1100	512	229	93	0	133	0	0 (-)
	spice*	115013	111	13615	14734	7498	2817	2614	343	555	20	21 (6.2)
	tomcatv	1815	1	170	189	95	34	20	0	20	0	0 (-)
SPECint92	compress	4302	16	762	791	435	165	130	17	16	0	1 (10)
	eqntott	14319	62	2527	2691	1396	577	367	61	125	0	12 (7.5)
	espresso	74277	361	15246	15874	8314	3170	2690	197	711	0	6 (12.8)
	li	31868	357	6141	6132	3274	1118	1270	149	122	0	10 (7.6)
	sc	44198	151	7544	7892	4157	1554	1460	161	222	0	22 (18.3)
other	lloops	3142	2	318	367	160	53	52	0	51	0	0 (-)

* One spice procedure not processed at this time.

structured code can arise from `goto` and `break` statements, and from returns from within a procedure. Many of the programs are call-intensive, suggesting that inlining and inter-procedural analysis could provide considerable benefit. The conditional nesting depth can be larger than the nesting of `if` statements. The body of all loops that have an exit test at their entry point are control dependent on the exit *Predicate*. Multi-way branches are fairly common in integer code. The last column of Table 1 shows that the average number of targets ranges from 4 to 18. The number of PDG nodes, arcs, and labels are an indication of the memory requirements of the representation. The average number of PDG nodes per basic block is approximately 1.7.

5.2. Node Labeling Statistics

A node's label length is equal to its conditional nesting depth. Table 2 shows that it is largest for benchmarks with unstructured and irregular code, like spice, sc, and li. The cases with the largest average and maximum number of labels arise in parsers with switch statements whose cases have returns in them. Because the returns create a path from the multi-way branch to the exit, all the paths from that split are not rejoined until the procedure exit node.

The time for CFG labeling, PDG construction, and label comparisons depends primarily on the length and number of labels at each node. The complexity of label comparisons is O(CU^2), where C is the conditional nesting depth, and U is the number of labels per node. Since the average and maximum values of these parameters shown in Table 2 are small, label comparisons are generally efficient. Table 2 also shows labeling and PDG construction time, in seconds, on a 133 MHz Alpha 3000/400. This time is most affected by the number of PDG nodes, the number of labels per node conditional nesting depth, and loop nesting depth. The implementation has yet to be optimized.

Table 2: Control Structure and Labeling Statistics, Ordered by Number of PDG Nodes

Benchmark name	Number of basic blocks	Avg. instr. per basic block	Number of PDG nodes	Avg. cond. nesting depth	Max cond. nesting depth	Avg # of labels per node	Max. # of labels per node	Avg. loop nesting depth	Max. loop nesting depth	Labeling, PDG construction (s)
tomcatv	9[illegible]	7.78	170	1.92	4	1.00	1	1.52	3	0.91
lloops	164	9.99	318	1.01	2	1.00	1	0.69	3	2.37
compress	467	4.00	762	8.61	19	2.57	48	0.52	2	3.98
nasa7	548	[illegible]13	985	2.01	5	1.00	1	1.44	4	4.09
ear	1737	4.36	281[illegible]	2.74	18	1.14	16	0.36	3	6.27
eqntott	1765	4.11	2933	6.40	16	2.53	84	0.81	7	17.11
li	3493	3.16	5323	8.23	21	2.74	1024	0.25	2	21.61
sc	4459	3.70	7544	10.03	24	3.86	81	0.75	7	229.29
spice	6850	7.11	12237	8.07	23	3.56	178	1.18	7	3500.91
espresso	8331	4.03	14008	3.85	15	1.29	49	0.58	4	197.22

6. Scheduling Scopes

Several researchers [2,3,4,5,6,8,11,13] have proposed the PDG as a better representation than the CFG for use in code optimization. The size of the scope for code scheduling is a key factor in code optimization. The results in this section highlight the available scopes for code motion and scheduling using the PDG representation, and show how they compare with available scopes using the CFG representation. Since the data dependence analysis of PEDIGREE is not yet complete, data dependences and specific code motion techniques are not addressed here.

6.1. Scheduling Scope Without Speculation

A PDG explicitly represents all control dependences. *Region* node are used to identify a set of control-equivalent basic blocks. All control-equivalent basic blocks can be grouped into one scope for fine-grained scheduling (subject to data dependences). These basic blocks may be separated by potentially large distances in the CFG, making it difficult for CFG-based techniques to harvest such parallelism. Table 3 presents the average sizes of these con-

Table 3: Scope Size Without Speculation

Benchmarks		ear	nasa7	spice	tomcatv	compress	eqntott	espresso	li	sc	lloops	WHM
PDG	B.B.	4.8	9.0	7.3	22.8	2.6	3.1	5.2	4.2	3.2	90.8	5.1
	Instr	24.3	87.8	57.1	242.9	10.3	14.4	25.2	18.5	12.2	1038.6	23.9
CFG	B.B.	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	Instr.	5.4	9.8	8.4	8.0	4.3	5.0	4.6	4.1	4.0	10.2	5.6

trol-equivalent PDG scopes and non-speculative CFG scopes. It is assumed that code is not moved to blocks with lower execution frequency. For example, moving code from outside a conditional onto all paths of the conditional requires replication and results in code explosion. This assumption limits PDG scope sizes more than CFG scope sizes. The weighted harmonic mean of the number of control-equivalent basic blocks, across all benchmarks, is 5.1, and the corresponding number of instructions is 23.9. For comparison, the analogous average scope sizes that can be obtained using CFGs without speculation are 1.0 basic blocks and 5.6 instructions. The ability to identify control-equivalent basic blocks enables PDGs to increase the average scope size for fine-grained scheduling to over five times that of the CFGs. Potentially, this ability could significantly increase instruction-level parallelism and resultant performance. Actual increases will depend on data dependences.

The difference in scope sizes is primarily due to the intrinsic weakness of the CFG: not explicitly representing control dependence information. The control equivalence of basic blocks before and after a conditional (or called procedure) cannot be determined, in general, with traversal along a single path. Consequently, code motion techniques that traverse CFG paths would not be able to determine this equivalence and would need to conservatively assume that these two basic blocks are control dependent. Use of the CFG effectively induces "false" control dependences. The scope for fine-grained code motion is restricted by these false dependences. Control-equivalent basic blocks that are dispersed throughout the program can be easily identified using the PDG.

6.2. Scheduling Scope With Speculation

Speculative scheduling involves moving instructions across basic block boundaries, or to be more precise, moving instructions so that they are executed prior to the resolution of their control dependence conditions. To perform speculative code motion, a speculation scope is usually identified by grouping multiple basic blocks that are adjacent in the CFG. The degree of speculation is typically measured in terms of either the number of basic block boundaries crossed or the number of conditional branches crossed. The opportunity for code motion generally increases with a larger scope, which can be obtained with higher degrees of speculation.

Figure 5 presents the PDG and CFG speculation scope sizes for various degrees of speculation (*DOS*). Figure 5 presents corresponding scope sizes when speculation across loop back arcs is considered. Given a DOS of *N*, the

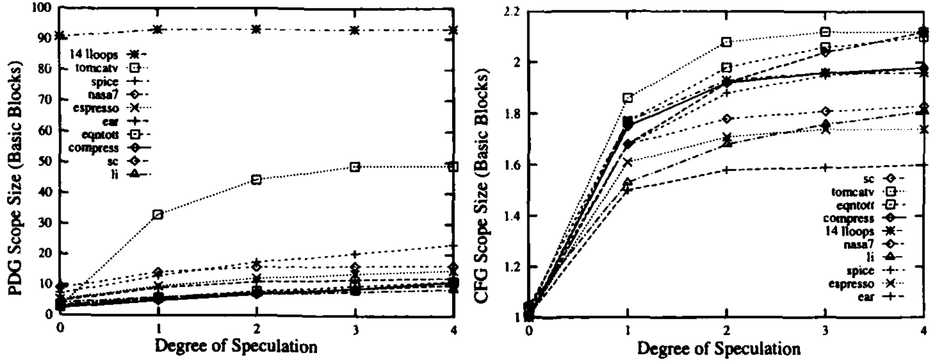


Figure 4. Scope size as a function of degree of speculation, for the PDG (a) and the CFG (b)

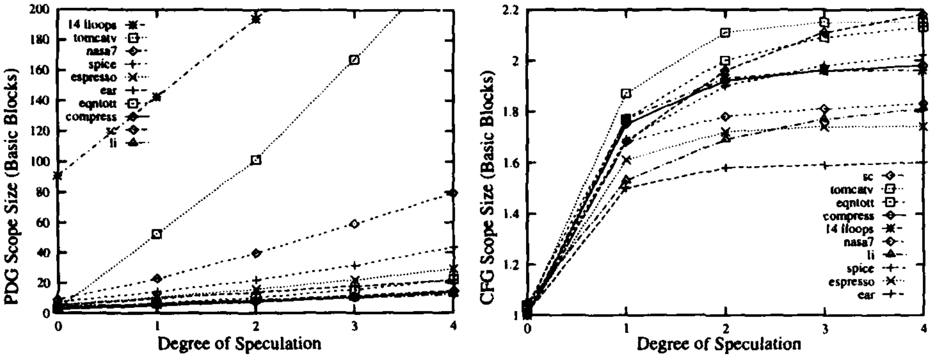


Figure 5. Scope size as a function of degree of speculation, considering speculation across loop back arcs, for the PDG (a) and the CFG (b).

CFG speculation scope size indicates the average number of basic blocks that are reachable from a reference basic block by traversing up to *N* divergences of control flow, i.e. *N* splits in the forward direction or *N* joins in the backward direction. CFG scopes are not permitted to extend past procedure calls. The PDG speculation scope is determined by crossing up to *N* true control dependences. Only control dependences are considered, not the original control flow order. Thus intervening *Call* nodes are ignored. Inlining should further increase PDG scope sizes.

The average PDG scope size in basic blocks across all benchmarks is 9.0 for DOS 1, and 14.7 for DOS 4. When speculation across loop back arcs is permitted, this number increases to 25.6 for DOS 4. The average scope size in instructions is 37.0 for DOS 1 and 57.7 for DOS 4. The corresponding CFG scope sizes are 1.7 (DOS 1) and 1.9 (DOS 4) basic blocks, and 8.4 (DOS 1) and 9.4 (DOS 4) instructions. PDG speculation scope size is 5.3 times greater on average than the CFG speculation scope size for DOS 1, and 7.7 times greater for DOS 4. Clearly, for speculative scheduling with a given degree of speculation, the PDG provides larger scopes for scheduling than the CFG. Furthermore, certain speculations in the CFG are not truly speculative, due to the false control dependences induced by the traversal of paths in a CFG. Consequently, some basic blocks in the speculation scope of degree *N* are not necessarily speculative of that degree.

Space does not permit presentation of more detailed data for individual code modules, but some insights are provided here. The bigger the procedure, the larger the PDG scopes, and the greater the benefits of the PDG vs. the CFG. For very small procedures without procedure calls, the PDG and CFG scope sizes are similar, although CFG scopes rise more slowly with DOS. The increase in scope size in the PDG levels off quickly for very shallow PDGs (e.g. espresso's cubestr) and more slowly for deeper ones (espresso's mv_reduce). Calls are very limiting to CFG scopes. Table 2 indicates that ear, spice, li and lloops have a high ratio of *Call* nodes to *Code* nodes. This may explain why these benchmarks have smaller CFG scope sizes. As can be observed from a comparison of Figure 5 and Figure 5, the relative contribution of speculation across loop back arcs is small for low DOS. For higher DOS, when other speculation potential is exhausted, consecutive loop iterations offer a continuous supply of parallelism. Loop speculation accounts for only 18% of the scope size increase of 4.7 basic blocks from DOS 0 to DOS 1. But it accounts for 77% of the scope size increase of 6.2 basic blocks from DOS 3 to DOS 4.

Two key differences between the CFG and the PDG are that the PDG represents the control dependences instead of the control flow of the CFG, and that the PDG has the capability for hierarchical representation using region nodes. The former removes the undesirable artifacts of the original sequential code. The latter allows encapsulation of information and more efficient traversal of the PDG during code motion. Initial results from the PEDIGREE tool appear to support the claims of the proponents of the PDG [2, 5, 6, 8, 13].

7. Summary and Potential PDG-Based Optimizations using PEDIGREE

The PDG described here is useful in exposing hierarchical control dependence structure. This information can be used to determine the kind of transformations that should be applied to each section of the program, e.g. speculation, loop optimization, and replication. PDG analysis suggests the benefits of increased scope size for different degrees of speculation, and can provide guidance in selecting appropriate aggressiveness for each procedure. The use of accurate control dependence information to avoid unnecessary speculation provides a 5x increase in scope size. Aggressive scheduling techniques are needed to exploit parallelism for non-loop code on wide architectures. Exploiting even a small fraction of the additional parallelism discussed here is a significant step toward that goal.

The existing PDG-based tool, PEDIGREE, provides a framework for future development of a number of useful capabilities. It can be extended to take advantage of PDG features to 1) increase the scheduling scope and guide speculation with accurate control dependence information, 2) perform inlining and inter-procedural analysis, 3) move any region of code along an unstructured path, 4) schedule for multiple-instruction stream architectures, and 5) perform binary optimization. The data from Section 6 suggests that the PDG scopes for code motion for a given degree of speculation are significantly larger than those of a CFG. Fine-grained code scheduling techniques can be implemented in PEDIGREE to take advantage of the larger scope. The *Call* and *Proc* nodes added here facilitate inlining, which can further increase scope size. They can also be used to summarize parallelism, variable usage, and execution frequency, in order to inform a global guidance layer of the benefits of inlining, and to facilitate interprocedural analysis. The implementation of inlining and use of branch profiling data will be completed shortly.

The PDG's features provide a solid basis for partitioning and scheduling code for multiple-instruction stream architectures. Such features include its ability to represent parallel schedules, its summary of parallelism information that guides trade-offs in the redistribution of parallelism, and its support for generalized multi-grained code motion. Parallelism studies [10] indicate that exploiting control parallelism in multiple instruction streams can lead to a three-fold performance increase over a single instruction stream for non-loop-intensive code.

The post-pass PEDIGREE tool currently parses assembly files and constructs the PDG for a given architecture specified in an architecture description file. Using this feature, PEDIGREE can be easily retargeted for other assembly languages. There is potential for using this tool to perform translation and optimization on existing object code. This will permit the effective execution of a large volume of existing software on new platforms without recompilation from the source. Future exploration of this potential application using PEDIGREE is planned.

References

[1] A. Aho, R. Sethi and J. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[2] V. Allan, J. Janardhan, R. Lee and M. Srinivas, "Enhanced Region Scheduling on a Program Dependence Graph", In *Proc. MICRO 25*, 1992, 72-80.

[3] R. Ballance, A. Maccabe and K. Ottenstein, "The Program Dependence Web: A Representation Supporting Control-, Data, and Demand-Driven Interpretation of Imperative Languages", In *Proc. PLDI*, 1990, 257-271.

[4] W. Baxter and H. Bauer III, "The Program Dependence Graph in Vectorization", In *Proc. PoPL*, 1989, 1-11.

[5] D. Bernstein and M. Rodeh, "Global Instruction Scheduling for Superscalar Machines", In *Proc. PLDI*, 1991, 241-255.

[6] J. Ferrante, K. Ottenstein and J.Warren. "The Program Dependence Graph and Its Use in Optimization", *ACM TOPLAS* 9(3), July 1987, 319–349.

[7] M.Girkar and C. Polychronopoulos, "Automatic Extraction of Functional Parallelism from Ordinary Programs," IEEE Transactions on Parallel and Distributed Systems, 3(2), March 1992, 166-178.

[8] R.Gupta and M. L. Soffa. "Region Scheduling", In *Proc. Second Int'l Conf. on Supercomputing*, 1987, 141–8.

[9] S. Horwitz and T. Reps, "The Use of Program Dependence Graphs in Software Engineering", In *Proc. International Conference on Software Engineering*, Melbourne Australia, 1992, 392-411.

[10] M. Lam and R. Wilson, "Limits of Control Flow on Parallelism", In *Proc. ISCA*, 1992, 46-57.

[11] C. Norris and L. Pollock, "Register Allocation over the Program Dependence Graph," In *Proc. PLDI*, 1994.

[12] K.Ottenstein and S. Ellcey, "Experience Compiling FORTRAN to Program Dependence Graphs", Software — Practice and Experience, 22(1), 1992, 41-62.

[13] V. Sarkar, "Automatic Partitioning of a Program Dependence Graph into Parallel Tasks", IBM Journal of Research and Development, 35(5/6), 1991, 779-804.

PART VI

COMPILING FOR PARALLEL MACHINES

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Static Analysis of Barrier Synchronization in Explicitly Parallel Programs

Tor E. Jeremiassen and Susan J. Eggers

Department of Computer Science and Engineering FR-35, University of Washington, Seattle, Washington 98195

Abstract: Many coarse-grained, explicitly parallel programs execute in phases delimited by barriers to preserve sets of cross process data dependencies. One of the major obstacles to optimizing these programs is the necessity to conservatively assume that any two statements in the program may execute concurrently. Consequently, compilers fail to take advantage of opportunities to apply optimizing transformations, particularly those designed to improve data locality, both within and across the phases of the program.

We present a simple and efficient compile time algorithm that uses the presence of barriers to perform non-concurrency analysis on coarse-grain, explicitly parallel programs. It works by dividing the program into a set of phases and computing the control flow between them. Each phase consists of one or more sequences of program statements that are delimited by barrier synchronization events and can execute concurrently. We show that the algorithm performs perfectly on all but one of our benchmarks.

Keyword Codes: D.1.3; I.1.3
Keywords: Concurrent Programming; Languages and Systems

1 Introduction

On cache coherent shared memory multiprocessors, much of the "unnecessary" communication, i.e., that which could be eliminated with locality enhancing optimizations, is coherency overhead caused by false sharing [22, 10]. False sharing occurs when multiple processors access different words in the same cache block. Although they do not actually share data, they incur its costs, because coherency operations are cache block-based. In a write-invalidate coherency protocol the overhead of false sharing takes the form of additional invalidations when a processor updates data and invalidation misses when other processors reread (different) data that reside in the invalidated cache block. In some coarse-grain, explicitly parallel applications, misses due to false sharing make up between 40% and 90% of all cache misses (over block sizes ranging from 8 bytes to 256 bytes) [10].

False sharing is caused by a mismatch between the memory layout of write-shared data and the cross-processor memory reference pattern to it. Manually changing the placement of this data to better conform to the memory reference pattern reduced false sharing misses by 40% to 75% [22, 10]. However, manual restructuring requires that the programmer pinpoint the data structures that suffer from false sharing in a particular memory (cache) architecture. This is hard to determine; knowledge of how each data object is shared is often non-intuitive, and each application must be tailored to the particular memory architecture of the system on which it is running.

For these reasons we have automated the elimination of false sharing. We have added a series of compiler-directed algorithms and a suite of transformations to a source-to-source restructurer to transform shared data at compile time. Our algorithms analyze explicitly parallel programs, producing information about their cross-processor memory reference patterns that identifies data structures susceptible to false sharing, and then chooses appropriate transformations. They were more successful than the programmer-directed approach in restructuring shared data, eliminating up to 97% of all false sharing misses.

The compiler analysis involves three separate stages. The first determines which sections of code each process executes by computing its control flow graph [13]. The second performs non-concurrency analysis [16] by examining the barrier synchronization pattern of the program, and delineating the program into phases that cannot execute in parallel The third stage performs an enhanced interprocedural, flow-insensitive, summary side-effect analysis [7] and static profiling [23] on a per-process basis (based on the control flow determined in stage one) for each phase (determined in stage two). The second stage of this process, the barrier synchronization analysis, is the subject of this paper.

Analyzing a program based on data summarized over the whole program often fails to recognize shifts in the reference pattern to shared data between different (non-concurrent) phases of the program. The barrier synchronization analysis algorithm addresses this problem by dividing the program into a set of sequentially executing phases, delineated by barriers, and computing the flow of control among them. Using the barrier analysis algorithm to obtain more detailed, per-phase summary information for shared data can be used, along with static profiling, to detect a dominant sharing pattern in the program and restructure for that pattern. For example, in one application in our workload the barrier analysis algorithm revealed that shared structures were accessed on a distinct per-process basis in all parts of the program except during the final convergence testing. Our algorithm therefore decided to transform the data by process, according to its dominant usage. Applying the barrier analysis to the program reduced the false sharing miss rate by 96% (on average, across multiple block sizes). Excluding it produced a value of only 8% for the same metric.

In the next section we describe our model of parallel programming. Section 3 places the barrier synchronization algorithm in context by summarizing the algorithms and transformations in our compiler. Section 4 describes the barrier algorithm in detail. Section 5 briefly describes the methodology and workload. Section 6 presents results of using the barrier algorithm: its ability to detect phases in all programs and its effect on eliminating false sharing in one. Section 7 discusses related work, and Section 8 concludes.

2 Model of Parallel Programming

Our compiler is aimed at coarse-grained, explicitly parallel programs for shared memory multiprocessors, similar to those found in the Stanford SPLASH application suite [20]. The granularity of parallelism in these programs is coarse, on the level of an entire process.

```
private int pid;
shared barrier_t MyBarr1, MyBarr2, MyBarr3;
shared int NumProcs;
:
for ( pid = 1; pid < NumProcs; pid++)
    if (fork() == 0) {
        Work();
        exit(0);
    }
MasterWork();
```

```
Work() {
    while (converged != 0) {
         SubPart1(pid);
s1:      Wait_Barrier(&MyBarr1);
         SubPart2(pid);
s2:      Wait_Barrier(&MyBarr2);
s3:      Wait_Barrier(&MyBarr3);

    }
```

```
MasterWork() {
    while (converged != 0) {
         SubPart1(pid);
s4:      Wait_Barrier(&MyBarr1);
         SubPart2(pid);
s5:      Wait_Barrier(&MyBarr2);
         converged = TestConverged();
s6:      Wait_Barrier(&MyBarr3);
    }
```

Figure 1: Example program segments.

In our model the number of processes equals the number of processors and processes do not migrate. The programs conform to an SPMD model of parallel programming: the processes all have identical code, but need not take the same path through the program. They may or may not access different shared data. Processes are created explicitly, e.g., using a *fork()* system call (illustrated in Figure 1). Processes are differentiated by the values of private variables (*pid* in Figure 1 is an example) or system calls that return process identifiers.

Process synchronization is performed using global barriers. When the control flow of a process reaches a barrier, it must wait until the other participating processes also reach the barrier. A barrier is global if all the processes in the program participate in the barrier. On bus-based multiprocessors barriers are commonly implemented using a structure that contains a shared counter [3]; we call this structure the *barrier variable*. Upon reaching the barrier, each process increments the counter and waits until the counter has reached the preset limit, at which time the counter is reset and the processes continue execution.

3 Enhancing Locality

To determine which data structures in a program are susceptible to false sharing, where locality may be improved and which transformations to apply at compile time, we analyze the program and compute an approximation of the memory accesses of each of its processes. To provide a context for the barrier analysis algorithm, that process is briefly outlined in the remainder of this section.

The first stage of the analysis is to determine the set of possible control flow paths through the program that each process may take. The most conservative assumption would be that every process may take any path through the program. However, in many cases this vastly overestimates the code that the individual processes may execute and consequently leads to overly conservative approximations of the processes' memory accesses. We use information about how the values of private variables, such as *pid* in Figure 1, vary across the processes to prune sections of the control flow graph that cannot be executed by the different processes. Complete details of this analysis may be found in [13].

The second stage consists of the barrier synchronization analysis algorithm. It divides the program into a set of non-concurrent phases, so that the sharing pattern for each phase may be analyzed separately. We discuss the details of this algorithm in section 4.

The third stage of the analysis uses the results from the first two stages to perform an interprocedural, flow-insensitive, summary side-effect analysis of the code each process executes in each phase of the program. The side-effect information includes regular section descriptors [11] to describe the sections of each array that each process accesses [1]. We have enhanced the summary side-effect analysis in two ways. First, we use static profiling [23] to produce a weighting of the side-effects with respect to estimated execution frequency. The weighting makes it easier to pinpoint which data structures suffer from false sharing by eliminating from consideration those that are accessed infrequently. Second, instead of merging all regular section descriptors for an array into a single descriptor in the summary information [4, 11], we only merge descriptors when very little or no information will be lost, or when the number of descriptors for a single array exceeds some small preset limit. Multiple regular section descriptors allow us to factor out irregular array access patterns that occur seldom and do not contribute to false sharing.

In order to eliminate or significantly reduce false sharing misses, data must be restructured so that (1) data that is only, or overwhelmingly, accessed by one processor is grouped together, and (2) write-shared data objects with no processor locality [1] do not share cache lines. We apply three different kinds of transformations, depending on the outcome of the static analysis. The first transformation groups data physically together by changing the layout of the target data structures in memory. When it is not possible to physically change the data layout (because, for example, the affected per-process data structure is embedded into the elements of a dynamically allocated list or graph data structure), we can achieve a similar result using the second transformation. It allocates data areas of memory for each processor, places shared data into them pointed to by pointers that replace the shared data in the target data structures. Although this adds a memory access to each reference of the affected data structures, the accesses are much more likely to be cache hits rather than invalidation misses. Thus the total amount of time spent to access the data is less. The third transformation uses padding to prevent affected data items from residing in the same cache block.

4 Barrier Synchronization Analysis

We define a *process segment* as the set of statement sequences along all barrier free paths in the control flow graph that start at one barrier synchronization call site i using barrier variable b_x, which we denote S_{i,b_x}, and end at another (possibly the same) barrier synchronization call site S_{j,b_y}. A process segment is identified by the barrier synchronization call sites that delimit it, i.e., the above process segment would be denoted (S_{i,b_x}, S_{j,b_y}). A *phase* of a program is the set of process segments that may execute concurrently between two global barrier synchronization events. The goal of the barrier analysis algorithm is to divide the program into a set of process segments and partition them into a set of phases.

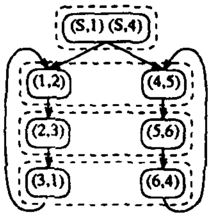
[1] A regular section descriptor is a vector of subscript positions in which each element describes the accessed portion of the array in that dimension as either an invariant expression, a range (giving invariant expressions for the lower bound, upper bound and stride), or unknown.

To help explain the algorithm we introduce the notion of a *barrier synchronization graph* $G_B = (V_B, E_B)$ for a program. It is a rooted, directed graph, defined as follows:

1. Every vertex $v \in V_B$ corresponds to the set of process segments $T_{v,b}$ that originate at the same barrier synchronization call site and terminate at barrier synchronization call sites that pass the barrier variable b. A dummy start node[2] (root vertex) is inserted to point to the vertices that contain process segments originating from a conceptual barrier at the beginning of the program. We do not insert an *end barrier*, but instead observe that the sections of the program not covered by process segments make up a separate program phase that ends in program termination.

2. There exists a directed edge $(u, v) \in E_B$, if there are process segments $(S_{i,b_x}, S_{j,b_y}) \in T_{u,b_y}$ and $(S_{m,b_y}, S_{n,b_z}) \in T_{v,b_z}$ such that $j = m$, i.e., S_{j,b_y} and S_{m,b_y} denote the same barrier synchronization call site.

Intuitively, the nodes in the gra;h represent sets of statement sequences of the program that do not contain barriers (process segments), and the edges represent barrier synchronization events that transfer control flow from one segment to another. The phases of the program can then be represented as a partition of the vertices all of whose segments can execute in parallel.



	Initially	Step 1	Step 2	Final Set of Phases
Phase A:	{(S,1),(S,4)}	{(S,1),(S,4)}	{(S,1),(S,4)}	{(S,1),(S,4)}
Phase B:	{(1,2)}	{(1,2),(4,5)}	{(1,2),(4,5)}	{(1,2),(4,5)}
Phase C:	{(2,3)}	{(2,3)}	{(2,3),(5,6)}	{(2,3),(5,6)}
Phase D:	{(3,1)}	{(3,1)}	{(3,1)}	{(3,1),(6,4)}
Phase E:	{(4,5)}			
Phase F:	{(5,6)}	{(5,6)}		
Phase G:	{(6,4)}	{(6,4)}	{(6,4)}	

Process Segments = {(S,1),(S,2),(1,2),(2,3),(3,1),(4,5),(5,6),(6,4)}

Figure 2: Barrier synchronization graph for program in Figure 1. Dashed boxes illustrate phases. Barrier variables have been left out for reasons of readability.

The results of the barrier synchronization analysis are conservative. If two process segments are in different phases, they do not execute concurrently. However, the converse may not be true, i.e., if two process segments are in the same phase, they may, but need not, execute in parallel.

4.1 Computing the Process Segments

The first stage of the barrier analysis algorithm divides the program into a set of process segments. For each barrier synchronization call site we compute which other sites can be reached along barrier free paths in the program. Conceptually we create a variable $SynchVar_n$ for each barrier synchronization call site S_n (as opposed to the barrier variable b_x that is passed as a parameter at S_n). We then treat each barrier synchronization call at site S_n as the use of its variable $SynchVar_n$ followed by the definition of the variables $SynchVar_i$ for all i. The problem reduces to that of computing which $SynchVar_i$, are live at the end of each barrier synchronization call site. (Recall that a variable is live at a point

[2] For simplicity, this node is not shown in any illustrations of the barrier synchronization graph.

in a program if its value can be used before it is redefined [2].) Since these conceptual variables cannot be aliased, an interprocedural solution to this live *barrier* problem is possible and tractable [18]. Live barriers can be formulated as a simple distributive data flow analysis problem [14], for which efficient solutions are known. To speed up computation, procedures that do not contain any barriers and that do not directly or indirectly call any procedures containing barriers can be summarized into single nodes in the program control flow graph (thus avoiding propagating information through each statement of these procedures) without affecting the accuracy of the solution.

Following the solution to the interprocedural *live barrier* problem, the process segments are computed by considering each barrier S_{i,b_x} in turn. For each barrier S_{j,b_y} that is live just after S_{i,b_x}, we create the process segment (S_{i,b_x}, S_{j,b_y}).

4.2 Partitioning Process Segments into Phases

The second stage of the barrier analysis algorithm partitions the process segments into sequentially executing phases. It first optimistically assumes that only process segments that start at the same barrier and end at barriers that pass the same barrier variable can execute concurrently. That is, each barrier synchronization call site S_{i,x_1} gives rise to phases: $((S_{i,x_1}, S_{j_1,y_1}), .., (S_{i,x_1}, S_{j_n,y_1}))$, $((S_{i,x_1}, S_{k_1,y_2}), .., (S_{i,x_1}, S_{k_n,y_2}))$ and $((S_{i,x_1}, S_{l_1,y_n}), .., (S_{i,x_1}, S_{l_n,y_n}))$, where $S_{j_1,y_1}, .., S_{j_n,y_1}$, $S_{k_1,y_2}, .., S_{k_n,y_2}$, and $S_{l_1,y_n}, .., S_{l_n,y_n}$ are barrier synchronization calls live at S_{i,x_1}. These initial phases correspond to the vertices of the barrier synchronization graph. Using a work queue approach, the phases are then selectively merged until no two process segments that may execute in parallel are contained in different phases.

Initially all phases $\mathcal{P}_i$ that contain multiple process segments are put on the work queue. Each phase $\mathcal{P}_i$ on the queue is then examined in turn. For each pair of process segments $(S_{j,b_x}, S_{k,b_y}), (S_{r,b_x}, S_{s,b_y}) \in \mathcal{P}_i$, we merge all phases that contain either S_{k,b_y} or S_{s,b_y} as the first barrier in any of their process segments and whose process segments all end in barrier call sites that pass the same barrier variable. The resulting program phase is added to the work queue. This continues until the work queue is empty, after which the final set of phases remain. Phases that contain process segments that terminate at barrier call sites that pass different barrier variables are never merged.

Figure 2 illustrates how the algorithm works for the program shown in Figure 1. Since the program has seven barriers (six plus the start barrier), the process segments are initially partitioned into seven phases, A through G. Initially, only phase A is put on the work queue, since it is the only phase that contains more than one process segment, specifically the process segments $(S, 1)$ and $(S, 4)$. As can be seen from the figure, barriers 1 and 4 are first components of process segments in phases B and E, respectively. Since they both end at barrier call sites that pass the same barrier variable, they are merged together in B, which is then added to the work queue. The presence of process segments $(1, 2)$ and $(4, 5)$ in B causes phase C and phase F to be merged. Lastly, phase D and G are merged to produce the final set of phases.

5 Methodology

The barrier analysis algorithm and our false sharing detection and restructuring algorithms have been implemented as separate passes in Parafrase-2 [19]. False sharing reductions were measured using trace-driven simulation of a bus-based, shared memory architecture. Execution times were measured on a 56-processor Kendall Square Research KSR1 [15], which has a cache block size of 128 bytes for the purpose of cache coherency.

The workload consisted of five programs, all written in C (Table 1). Water, LocusRoute and Mp3d are from the Stanford SPLASH benchmarks [20]. Water evaluates the forces and potentials in a system of water molecules in liquid state; Mp3d solves a problem involving particle flow at extremely low density; and LocusRoute is a commercial quality VLSI standard cell router. Topopt [8] performs topological optimization on VLSI circuits using a parallel simulated annealing algorithm. Maxflow [6] computes the maximum flow in a directed graph. Mincut [9] partitions a graph using simulated annealing.

6 Results

We present two types of results. The first evaluates the effectiveness of the barrier analysis algorithm in defining phases and process segments by comparing the number of phases created by the algorithm to the actual dynamic synchronization behavior of each program. The second illustrates the usefulness of the algorithm by measuring the reduction in false sharing for one application.

Program	Lines of C	Barriers	Phases	Segments per phase	Segments Total
LocusRoute	6709	2	1	2	2
Maxflow	810	6	6	1, 1, 1, 1, 1, 1	6
Mincut	479	7	3	1, 4, 16	21
Mp3d	1653	6	7	1, 1, 1, 1, 1, 1, 1	7
Topopt	2206	10	5	4, 3, 2, 2, 4	15
Water	1451	6	8	1, 1, 1, 1, 1, 1, 1, 1	8

Table 1: Results of applying the barrier analysis algorithm to the benchmarks.

Table 1 summarizes the results of applying our algorithm to each benchmark. The results are given as the number of phases computed (excluding the termination phase), the number of process segments for each phase and the total number of process segments (again, excluding those in the termination phase). Figure 3 shows the barrier synchronization graphs for each of the benchmarks with the set of phases illustrated by dashed boxes. The algorithm correctly computes the set of process segments for all benchmark programs. In addition, it partitions these segments perfectly for all but one of the benchmarks. The results listed in Figure 3 and Table 1 show that the algorithm performs flawlessly in dividing LocusRoute (trivially), Maxflow, Mp3d, Topopt and Water into a maximum number of phases that coincide with the actual dynamic behavior of the programs. That is, for these programs the results are exact, not conservative.

For Mincut, the large number of cycles of vastly different lengths in the barrier synchro-

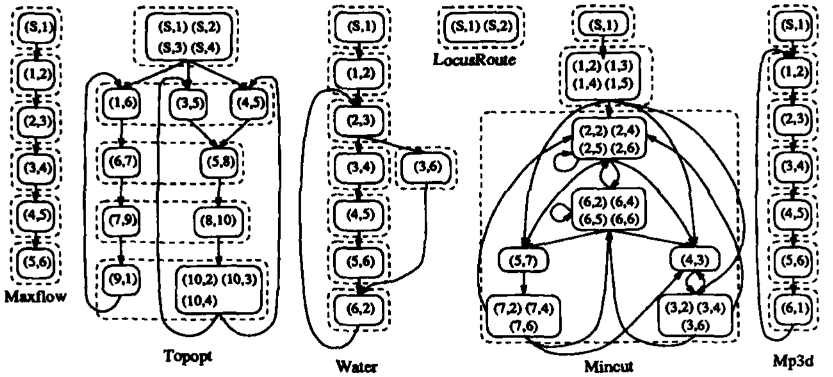


Figure 3: Program phases computed by the barrier synchronization analysis algorithm.

nization graph forces the algorithm to make a conservative approximation of the set of process segments that can execute concurrently. Because different processes may traverse different paths through the graph, specifically, different length cycles, they may get "out of synch" with one another. Thus, any two process segments in that phase could conceivably execute in parallel. It therefore places all process segments involved in these cycles into the same phase.

We use the barrier analysis algorithm in the context of a series of algorithms and data transformations designed to reduce false sharing. When different phases of a program exhibited different memory access patterns, barrier synchronization analysis was extremely effective in eliminating false sharing. Topopt illustrates this situation.

In the original, untransformed version of Topopt, false sharing caused 60% of all cache misses, averaged over block sizes between 8 and 256 bytes. Without barrier synchronization analysis shared data restructuring could only eliminate 8% of the false sharing misses. When barrier analysis was included, false sharing misses dropped an average of 96%. The consequence of eliminating false sharing misses and coherency operations was an improvement in execution time. When running on the KSR1, transformations reduced the execution time for Topopt by 15% when barrier synchronization analysis was included in the false sharing analysis, versus only 5% when it was not.

For the other benchmarks in our workload the impact was more modest. They fell into one of three categories. First, LocusRoute exhibited very little false sharing originally and therefore had very little room for improvement. Second, Mincut, Mp3d and Water suffered from more false sharing, but the analysis without the barrier analysis algorithm eliminated almost all of it. Third, for Maxflow the side-effect a·. ιysis concluded that the data structures should be left untransformed.

7 Related Work

Related work primarily concentrates on analyzing event variable synchronization to detect race conditions in parallel programs for the purpose of debugging, as opposed to compile time program optimization. Their analysis is mainly focused on event variable synchronization, i.e., `post` and `wait` or locks, and few show experimental results attesting to the effectiveness of their algorithms. Taylor [21] presents an algorithm that analyzes rendezvous in concurrent Ada programs and is exponential in the number of tasks in the program. The algorithm computes the set of possible concurrency states for a program and determines the possible sequences in which they may occur. McDowell [17], and Helmbold and McDowell [12] also enumerate possible concurrency states, but present techniques to reduce the number of states. The worst case number of states remains superpolynomial, however. Callahan, Kennedy and Subhlok [5] uses a synchronized control flow graph to determine when possible race conditions exist in parallel programs. Their model of parallel programming is based on parallel case and parallel do constructs, and their analysis is focused on loop nests. Our model of parallel programming is based on process level parallelism, and we perform our analysis over the entire program.

Masticola and Ryder [16] present a framework for non-concurrency analysis of Ada tasks. Using four refinement strategies they apply iteration to conservatively approximate which basic blocks cannot execute concurrently. Our algorithm has some similarity to their strategy of *pinning*; however, there are some key differences. First, they concentrate on pairwise process synchronization, while we address global process synchronization. Second, our solution is applicable to programs written under a different model of parallel programming. Finally, our algorithm is simpler and therefore more efficient.

8 Summary

We have presented a simple and efficient algorithm that statically analyzes the barrier synchronization pattern of coarse-grained, explicitly parallel programs. It uses a version of live variable analysis to divide each program up into a set of process segments, and by selectively merging vertices in the barrier synchronization graph, partitions these into non-concurrent program phases.

The algorithm performs well in computing the phases between barriers in parallel programs. Out of our six benchmarks, it produced a solution identical to the actual barrier synchronization pattern of each program in all but one case. For the last program the result was a conservative but safe approximation to the dynamic barrier synchronization pattern.

References

[1] A. Agarwal and A. Gupta. Memory-reference characteristics of multiprocessor applications under mach. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 215-225, May 1988.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1988.

[3] C.J. Beckmann and C.D. Polychronopoulos. The effect of barrier syncrhonization and scheduling overhead on parallel loops. In *International Conference on Parallel Processing*, volume II, pages 200-204, August 1989.

[4] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, (5):517–550, 1988.

[5] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of Event Synchronization in a Parallel Programming Tool. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 21–30, March 1990.

[6] F.J. Carrasco. A parallel maxflow implementation. CS411 Project Report, Stanford University, March 1988.

[7] K.D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Conference on Programming Languages Design and Implementation*, pages 57–66, June 1988.

[8] S. Devadas and A.R. Newton. Topological optimization of multiple level array logic. In *IEEE Transactions on Computer-Aided Design*, November 1987.

[9] J.A. Dykstal and T.C. Mowry. MINCUT: Graph partitioning using parallel simulated annealing. CS411 Project Report, Stanford University, March 1989.

[10] S.J. Eggers and T.E. Jeremiassen. Eliminating false sharing. In *International Conference on Parallel Processing*, volume I, pages 377–381, August 1991.

[11] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3), July 1991.

[12] D.P. Helmbold and C.E. McDowell. Computing reachable states of parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 76–84, May 1991.

[13] T.E. Jeremiassen and S.J. Eggers. Computing per-process summary side-effect information. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Fifth Workshop on Languages and Compilers for Parallelism*, August 1992.

[14] J. Kam and J. Ullman. Global data flow analysis and iterative algorithms. *JACM*, 23(1):159–171, January 1976.

[15] Kendall Square Research. *KSR-1 Principles of Operation*, 1992.

[16] S.P. Masticola and B.G. Ryder. Non-concurrency analysis. In *Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 129–138, May 1993.

[17] C.E. McDowell. A Practical Algorithm for Static Analysis of Parallel Programs. *Journal of Parallel and Distributed Computing*, 6:515–536, June 1989.

[18] E. Myers. A precise inter-procedural data flow algorithm. In *Symposium on Principles of Programming Languages*, pages 219–230, January 1981.

[19] C. Polychronopoulos, M. Girkar, M. Haghighat, C.L. Lee, B. Leung, and D. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *International Conference on Parallel Processing*, volume II, pages 39–48, August 1989.

[20] J.P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, 1991.

[21] R.N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, 1983.

[22] J. Torrellas, M.S. Lam, and J.L. Hennessy. Shared data placement optimizations to reduce multiprocessor cache miss rates. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II, pages 266–270, August 1990.

[23] D.W. Wall. Predicting program behavior using real or esitmated profiles. In *Conference on Programming Language Design and Implementation*, pages 59–70, June 1991.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)
1994 IFIP.

Exploiting the Parallelism Exposed by Partial Evaluation

R. Surati[a] and A. Berlin[b]

[a] MIT Artificial Intelligence Laboratory, 545 Technology Square, Cambridge, MA 02139, USA

[b] Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA

Abstract: We describe an approach to parallel compilation that seeks to harness the vast amount of fine-grain parallelism that is exposed through partial evaluation of numerically-intensive scientific programs. We have constructed a parallelizing compiler which uses partial evaluation to break down data abstractions and program structure, producing huge basic blocks that contain large amounts of fine-grain parallelism. To utilize this parallelism, we have developed a technique for automatically mapping the fine grain parallelism onto a coarser grain parallel computer architecture. We selectively group the fine-grain operations together so as to adjust the parallelism grain-size to match the inter-processor communication capabilities of the target architecture. On an important scientific problem, code produced by our compiler for the *Supercomputer Toolkit* parallel computer runs 6.2 times faster on eight processors than on one. For an important class of scientific applications, the coupling of partial evaluation with static scheduling techniques eliminates the need to require programmers to obscure programs by manually exposing the parallelism implicit in a computation.

Keyword Codes:
Keywords: Parallel Compilation; Partial Evaluation; Parallel Instruction Scheduling; Fine-grain Parallelism

1 Introduction

Previous work has shown that partial evaluation is good at bıeaking down data abstraction and exposing underlying fine-grain parallelism in a program [3]. We have written a novel compiler which couples partial evaluation with static scheduling techniques to exploit this fine-grain parallelism by automatically mapping it onto a coarse-grain parallel architecture.

Partial evaluation eliminates the barriers to parallel execution imposed by the data representation and the control structure of a program by taking advantage of information about the particular problem a program will be used to solve. For example, partial evaluation is able to perform at compile-time most data structure references, procedure calls, and conditional branches related to data structure size, leaving mostly numerical computations to be performed at run time. Partial evaluation is particularly effective on numerically-oriented scientific programs, since they tend to be mostly data-independent, meaning that they contain large regions in which the operations to be performed do not depend on the numerical values of the data being manipulated. For instance, matrix multiplication performs the same set of operations, regardless of the particular numerical values of the matrix elements. We use partial evaluation to produce huge basic blocks from these data-independent numerical regions. These basic blocks often contain thousands of instructions, two orders of magnitude larger than the basic blocks that typically arise in high-level language programs. To benefit from the fine-grain parallelism contained in these huge basic blocks, we schedule the partially-evaluated program for parallel execution primarily by performing the operations within an *individual* basic block in parallel.

In order to automatically map the freshly derived fine-grain parallelism onto a multiprocessor, we developed a technique which coarsens the dataflow graph by selectively aggregating operations together. This technique uses heuristics which take the communication bandwidth, inter-processor communication latency, and processor architecture all into consideration. High inter-processor communication latency requires that there be enough parallelism available to allow each processor to continue to initiate operations, even while waiting for results produced elsewhere to arrive. Limited communication bandwidth severely restricts the parallelism grain size that may be utilized by requiring that most values used by a processor be produced on that processor, rather than being received from another processor. Our approach addresses these problems by tailoring the grain size adjustment and scheduling heuristics to match the communication capabilities of the target architecture.

Our compiler operates in four major phases. The first phase performs partial evaluation, followed by traditional compiler optimizations, such as constant folding and dead-code elimination. The second phase analyzes locality constraints within each basic block, locating operations that depend so closely on one another that it is clearly desirable that they be computed on the same processor. These closely related operations are grouped together to form a higher grain size instruction, known as a *region*. The third compilation phase uses heuristic scheduling techniques to assign each region to a processor. The final phase schedules the individual operations for execution within each processor, accounting for pipelining, memory access restrictions, register allocation, and final allocation of the inter-processor communication pathways.

The target architecture of our compiler is the *Supercomputer Toolkit* , a parallel processor consisting of eight independent VLIW processors connected to each other by two shared communication busses [5]. Performance measurements of actual compiled programs running on the *Supercomputer Toolkit* show that the code produced by our compiler for an important astrophysics application[18] runs 6.2 times faster on an eight-processor system than does near-optimal code executing on a single processor. The compilation process of this real world application is used as an example throughout this paper.

2 The Partial Evaluator

Partial evaluation converts a high-level, abstractly written, general purpose program into a low-level program that is specialized for the particular application at hand. For instance, a program that computes force interactions among a system of N particles might be specialized to compute the gravitational interactions among 5 planets of our particular solar system. This specialization is achieved by performing in advance, at compile time, all operations that do not depend explicitly on the actual numerical values of the data.

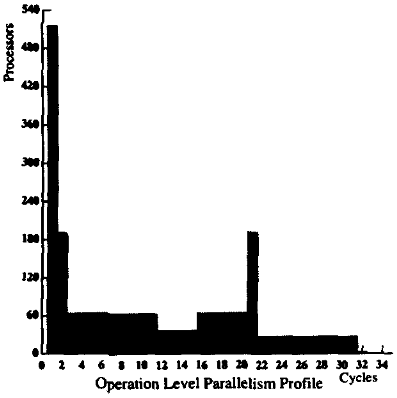


Figure 1: Parallelism profile of the 9-body problem. This graph represents all of the parallelism available in the problem, taking into account the varying latency of numerical operations.

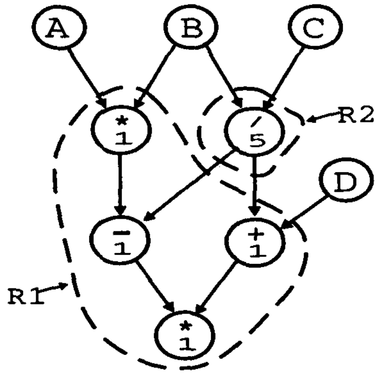


Figure 2: **A Simple Region Forming Heuristic.** A region is formed by grouping together operations that have a simple producer/consumer relationship. This process is invoked repeatedly, with the region growing in size as additional producers are added. The region-growing process terminates when no suitable producers remain, or when the maximum region size is reached. A producer is considered suitable to be included in a region if it produces its result solely for use by that region. (The numbers shown within each node reflect the computational latency of the operation.)

Region Size	Number of Regions
1	108
2	28
3	28
5	56
6	1
7	8
14	36
41	24
43	3

Table 1: The numerical operations in the 9-body program were divided into regions based on locality. This table shows how region size can vary depending on the locality structure of the computation. Region size is measured by computational latency (cycles). The program was divided into 292 regions, with an average region size of 7.56 cycles. The maximal region size used was 43 cycles

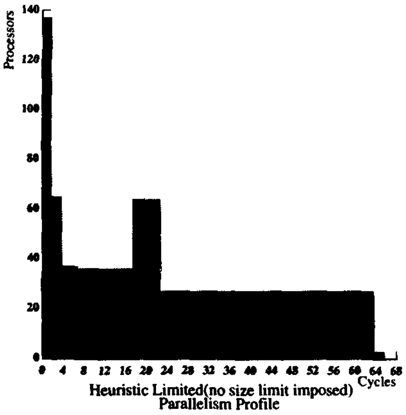


Figure 3: Parallelism profile of the 9-body problem after operations have been grouped together to form regions. Comparison with Figure 1 clearly shows that increasing the grain-size significantly reduced the opportunities for parallel execution. The maximum speedup factor dropped from 69 to 49 times faster than a single processor execution.

Many data structure references, procedure calls, conditional branches, table lookups, loop iterations, and even some numerical operations may be performed in advance, at *compile time*, leaving only the underlying numerical operations to be performed at run time

Our compiler exposes fine-grain parallelism using a simple partial evaluation strategy based on a symbolic execution technique described in [4, 3].[1] Despite this technique's simplicity, it works well at exposing fine-grain parallelism. Figure 1 illustrates a parallelism profile analysis of the nine-body gravitational attraction problem of the type discussed in [18].[2] Partial evaluation exposed so much low-level parallelism that in theory, parallel execution could speed up the computation by a factor of 69 over a uniprocessor.

3 Adjusting the Grain Size

Searching for an optimal schedule for a program which exploits fine-grain parallelism is both computationally expensive and difficult to achieve. Rather than do an exhaustive search for the optimal schedule, we developed a heuristic technique to coarsen the exposed fine-grain parallelism to a grain size suitable for critical-path based static scheduling. Prior to initiating critical-path based scheduling, we perform locality analysis that groups together operations that depend so closely on one other that it would not be practical to place them in different processors. Each group of closely interdependent operations forms a larger grain size macro-instruction, which we refer to as a *region*.[3] Some regions are large, while others may be as small as one fine-grain instruction. In essence, grouping operations together to form a region is a way of simplifying the scheduling process by deciding in advance that certain opportunities for parallel execution will be ignored due to limited communication capabilities.

Since operations within a region will occur on the same processor, the maximum region size must be chosen to match the communication capabilities of the target architecture. For instance, if regions are permitted to grow too large, a single region might encompass the entire data-flow graph, forcing the entire computation to be performed on a single processor! Although strict limits are therefore placed on the maximum size of a region, regions need not be of uniform size. Indeed, some regions will be large, corresponding to localized computation of intermediate results, while others will be quite small, corresponding to results that are used globally throughout the computation.

We have experimented with several different heuristics for grouping operations into regions. The optimal strategy for grouping instructions into regions varies with the application and with the communication limitations of the target architecture. However, we have found that even a relatively simple grain size adjustment strategy dramatically improves the performance of the scheduling process. As illustrated in Figure 2, when a value is used by only one instruction, the producer and consumer of that value may be grouped together to form a region, thereby ensuring that the scheduler will not place the

[1] More complex partial evaluation strategies that address data-dependent computations may be found in [9, 11, 10].

[2] Specifically, one time-step of a 12th-order Stormer integration of the gravity-induced motion of a 9-body solar system.

[3] The name *region* was chosen because we think of the grain size adjustment technique as identifying "regions" of locality within the data-flow graph. The process of grain size adjustment is closely related to the problem of graph multisection, although our region-finder is somewhat more particular about the properties (shape, size, and connectivity) of each "region" sub-graph than are typical graph multisection algorithms.

producer and consumer on different processors in an attempt to use spare cycles wherever they happened to be available. Provided that the maximum region size is chosen appropriately,[4] grouping operations together based on locality prevents the scheduler from making gratuitous use of the communication channels, forcing it to focus on scheduling options that make more effective use of the limited communication bandwidth.

An important aspect of grain size adjustment is that the grain size is not increased uniformly. As shown in Table 1, some regions are much larger than others. Indeed, it is important not to forcibly group non-localized operations into regions simply to increase the grain size. For example, it is likely that the result produced by an instruction that has many consumers will be transmitted amongst the processors, since it is not practical to place all of the consumers on the result-producing processor. In this case, creating a large region by grouping together the producer with only some of the consumers increases the grain size, but does not reduce inter-processor communication, since the result would need to be transmitted anyway. In other words, it only makes sense to limit the scheduler's options by grouping operations together when doing so will clearly reduce inter-processor communication.

4 Parallel Scheduling

Exploiting locality by grouping operations into regions forces closely-related operations to occur on the same processor. Although this reduces inter-processor communication requirements, it also eliminates many opportunities for parallel execution. Figure 3 shows the parallelism remaining in the 9-body problem after operations have been grouped into regions. Comparison with Figure 1 shows that increasing the grain size eliminates about half of the opportunities for parallel execution. The challenge facing the parallel scheduler is to make effective use of the limited parallelism that remains, while taking into consideration such factors as communication latency, memory traffic, pipeline delays, and allocation of resources such as processor buses and inter-processor communication channels.

Our compiler schedules operations for parallel execution in two phases. The first phase, known as the region-level scheduler, is primarily concerned with coarse-grain assignment of regions to processors, generating a rough outline of what the final program will look like. The region-level scheduler assigns each region to a processor; determines the source, destinations, and approximate time of transmission of each inter-processor message; and determines the preferred order of execution of the regions assigned to each processor. The region-level scheduler takes into account the latency of numerical operations, the inter-processor communication capabilities of the target architecture, the structure (critical path) of the computation, and which data values each processor will store in its memory. The region-level scheduler does *not* concern itself with finer-grain details such as the pipeline structure of the processors, the detailed allocation of each communication channel, or the ordering of individual operations within a processor. At the coarse grain size associated with the scheduling of regions, a straightforward set of critical-path based scheduling heuristics[5] have proven quite effective. For the 9-body problem example, the

[4] The region size must be chosen such that the computational latency of the operations grouped together is well-matched to the communication bandwidth limitations of the architecture. If the regions are made too large, communication bandwidth will be under utilized since the operations within a region do not transmit their results.

[5] The heuristics used by the region-level scheduler are closely related to list-scheduling [13]. A detailed discussion of the heuristics used by the region-level scheduler is presented in [1].

computational load was spread so evenly that the variation in utilization efficiency among the 8 processors was only one percent.

The final phase of the compilation process is instruction-level scheduling. The region-level scheduler provides the instruction-level scheduler with an ordered list of regions to execute on each processor along with a list of results that need to be transmitted when they are computed. The instruction-level scheduler chooses the final ordering of low-level operations within each processor, taking into account processor pipelining, register allocation, memory access restrictions, and availability of inter-processor-communication channels. Whenever possible, the order of operations is chosen so as to match the preferences of the region-level scheduler, represented by the ordered list of regions. However, the instruction-level scheduler is free to reorder operations as needed, intertwining operations among the regions assigned to a particular processor, without regard to which coarse-grain region they were originally a member of. This strategy allows the instruction scheduler to maintain a schedule similar to the one suggested by the region scheduler, thereby ensuring that the results will be produced at approximately the time that other processors are expecting them, while still taking advantage of fine grain parallelism available in other regions to fill pipeline slots as needed.

The instruction-level scheduler derives low-level pipelined instructions for each processor, choosing the exact time and communication channel for each inter-processor transmission, and determining where values will be stored within each processor. The instruction-level scheduling process begins with a data-use analysis that determines which instructions share data values and should therefore be placed near each other for register allocation purposes. This data-use information is combined with the higher-level ordering preferences expressed by the region-level scheduler, producing a scheduling priority for each instruction. The instruction scheduling process then proceeds one cycle at a time, performing scheduling of that cycle on *all* processors before moving on to the next cycle. Instructions compete for resources based on their scheduling priority; in each cycle, the highest-priority operation whose data and processor resources are available will be scheduled. This competition for data and resources helps to keep each processor busy, by scheduling low-priority operations whose resources are available whenever the resources for higher priority computations are not available. Indeed, when the performance of the instruction-scheduler is measured independently of the region-level scheduler, by generating code for a single *Supercomputer Toolkit* VLIW processor, utilization efficiencies in excess of 99.7% are routinely achieved, representing nearly optimal code.

An aspect of the scheduler that has proven to be particularly important is the retroactive scheduling of memory references. Although computation instructions (such as + or *) are scheduled on a cycle-by-cycle basis, memory LOAD instructions are scheduled retroactively, wherever they happen to fit in. For instance, when a computation instruction requires that a value be loaded into a register from memory, the actual memory access operation[6] is scheduled in the past for the earliest moment at which both a register and a memory-bus cycle are available; the memory operation may occur fifty or even one-hundred instructions earlier than the computation instruction. *Supercomputer Toolkit* memory operations must compete for bus access with inter-processor messages, so retroactive scheduling of memory references helps to avoid interference between memory and communication traffic.

[6] On the toolkit architecture, two memory operations may occur in parallel with computation and address-generation operations. This ensures that retroactively scheduled memory accesses will not interfere with computations from previous cycles that have already been scheduled.

Program	Single Processor Cycles	Eight Processors Cycles	Speedup
ST6	5811	954	6.1
ST9	11042	1785	6.2
ST12	18588	3095	6.0
RK9	6329	1228	5.2

Table 2: Speedups of various applications running on 8 processors. Four different computations have been compiled in order to measure the performance of the compiler: a 6 particle stormer integration(ST6), a 9 particle stormer integration(ST9), a 12 particle stormer integration(ST12), and a 9 particle fourth-order Runge Kutta integration(RK9). Speedup is the single processor execution time of the computation divided by the total execution time on the multiprocessor.

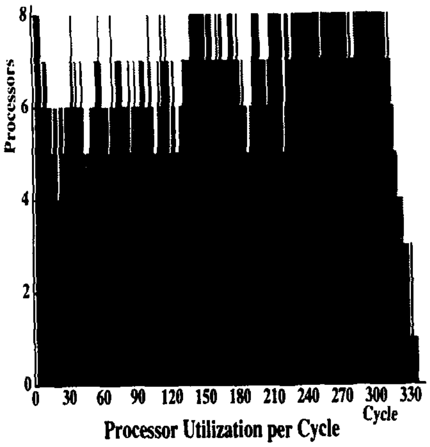


Figure 4: The result of scheduling the 9-body problem onto 8 *Supercomputer Toolkit* processors. Comparison with with the region-level parallelism profile (figure 3) illustrates how the scheduler spread the course-grain parallelism across the processors. A total of 340 cycles are required to complete the computation. On average, 6.5 of the 8 processors are utilized during each cycle.

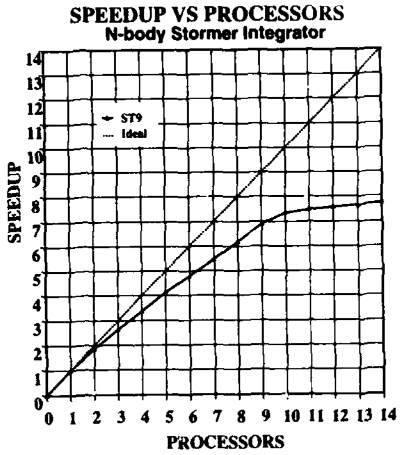


Figure 5: Speedup graph of Stormer integrations. Ample speedups are available to keep the 8-processor *Supercomputer Toolkit* busy. However, the incremental improvement of using more than 10 processors is relatively small.

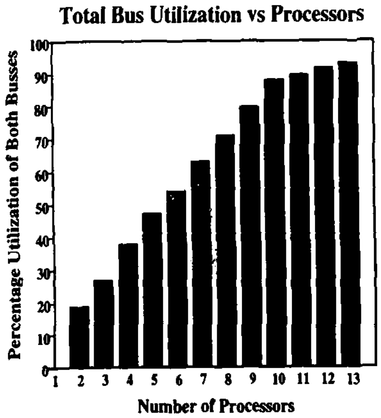


Figure 6: Utilization of the inter-processor communication pathways. The communication system becomes saturated at around 10 processors. This accounts for the lack of incremental improvement available from using more than 10 processors that was seen in Figure 5.

Figure 4 illustrates the effectiveness of the instruction level scheduler on the nine-body problem example.

5 Performance Measurements

The *Supercomputer Toolkit* and our associated compiler have been used for a wide variety of applications, ranging from computation of human genetic pedigrees to the simulation of electrical circuits. The applications that have generated the most interest from the scientific community involve various integrations of the N-body gravitational attraction problem.[7] Parallelization of these integrations has been previously studied by Miller[17], who parallelized the program by using *futures* to manually specify how parallel execution should be attained. Miller shows how one can re-write the N-body program so as to eliminate sequential data structure accesses to provide more effective parallel execution, manually performing some of the optimizations that partial evaluation provides automatically. Others have developed special-purpose hardware that parallelizes the 9-body problem by dedicating one processor per planet.[16] Previous work in partial evaluation [2, 4, 3] has shown that the 9-body problem contains large amounts of fine-grain parallelism, suggesting that more subtle parallelizations are possible without the need to dedicate one processor to each planet.

We have measured the effectiveness of coupling partial evaluation with grain size adjustment to generate code for the *Supercomputer Toolkit* parallel computer, an architecture that suffers from serious inter-processor communication latency and bandwidth limitations. Table 2 shows the parallel speedups achieved by our compiler for several different N-body interaction applications. Figure 5 focuses on the 9-body program (ST9) discussed earlier in this paper, illustrating how the parallel speedup varies with the number of processors used. Note that as the number of processors increases beyond 10, the speedup curves level off. A more detailed analysis has revealed that this is due to the saturation of the inter-processor communication pathways, as illustrated in Figure 6. The accuracy of these results was verified by executing the 9-body program on the actual *Supercomputer Toolkit* hardware in an eight processor configuration.

An important drawback to the partial evaluation approach is that it results in the unrolling of loops, which can potentially lead to an explosion in the size of the compiled program. We have found that depending on the size of the data set being manipulated, partial evaluation may reduce the overall size of the program, by eliminating data accesses, branches, and abstraction-manipulation code; or partial evaluation may increase the size of the program by iterating over a large data set. The key to making successful use of the partial evaluation technique is to not carry it too far. For relatively small applications, such as the 9-body integration program, it was practical to partially-evaluate the entire computation; on the other hand, if one was simulating a galaxy containing millions of stars, it would probably be best not to partially-evaluate some of the outermost loops! Our work focuses on achieving efficient parallel execution of the partially-evaluated segments of a program, leaving the decision of which portions of a program should be subjected to this compilation technique up to the programmer.

[7] For instance, [18] describes results obtained using the *Supercomputer Toolkit* that prove that the solar system's dynamics are chaotic.

6 Related Work

The use of partial evaluation to expose parallelism makes our approach to parallel compilation fundamentally different from the approaches taken by other compilers. Traditionally, compilers have maintained the data structures and control structure of the original program. For example, if the original program represents an object as a doubly-linked list of numbers, the compiled program would as well. Only through partial evaluation can the data structures used by the programmer to think about the problem be removed, leaving the compiler free to optimize the underlying numerical computation, unhindered by sequentially-accessed data structures and procedure calls. However, the drawback to the partial-evaluation approach is that it is only highly efffective for applications that are mostly data-independent.

Many compilers for high-performance architectures use program transformations to exploit low-level parallelism. For instance, compilers for vector machines unroll loops to help fill vector registers. Other parallelization techniques include trace-scheduling, software pipelining, vectorizing, as well as static and dynamic scheduling of data-flow graphs.

6.1 Trace Scheduling

Compilers that exploit fine-grain parallelism often employ trace-scheduling techniques [14] to guess which way a branch will go, allowing computations beyond the branch to occur in parallel with those that precede the branch. Our approach differs in that we use partial evaluation to take advantage of information about the specific application at hand, allowing us to totally eliminate many data-independent branches, producing basic blocks on the order of several thousands of instructions, rather than the ten to thirty instructions typically encountered by trace-scheduling based compilers. An interesting direction for future work would be to add trace-scheduling to our approach, to optimize across the data-dependent branches that occur at basic block boundaries.

Most trace-scheduling based compilers use a variant of list-scheduling[13] to parallelize operations within an individual basic block. Although list-scheduling using critical-path based heuristics is very effective when the grain size of the instructions is well-matched to inter-processor communication bandwidth, we have found that in the case of limited bandwidth, a grain size adjustment phase is required to make the list-scheduling approach effective.[8]

6.2 Software Pipelining

Software Pipelining [12] optimizes a particular fixed size loop structure such that several iterations of the loop are started on different processors at constant intervals of time. This

[8] The partial-evaluation phase of our compiler is currently not very well automated, requiring that the programmer provide the compiler with a set of input data structures for each data-independent code sequence, as if the data-independent sequences are separate programs being glued together by the data-dependent conditional branches. This manual interface to the partial evaluator is somewhat of an implementation quirk; there is no reason that it could not be more automated. Indeed, several *Supercomputer Toolkit* users have built code generation systems on top of our compiler that automatically generate complete programs, including data-dependent conditionals, invoking the partial evaluator to optimize the data-independent portions of the program. Recent work by Weise, Ruf, and Katz[9, 10] describes additional techniques for automating the partial-evaluation process across data-dependent branches.

increases the throughput of the computation. The effectiveness of software pipelining will be determined by whether the grain size of the parallelism expressed in the looping structure employed by the programmer matches the architecture: software pipelining can not parallelize a computation that has its parallelism hidden behind inherently sequential data references and spread across multiple loops. The partial-evaluation approach on such a loop structure would result in the loop being completely unrolled with all of the sequential data structure references removed and all of the fine grain parallelism in the loop's computation exposed and available for parallelization. In some applications, especially those involving partial differential equations, fully unrolling loops may generate prohibitively large programs. In these situations, partial evaluation could be used to optimize the innermost loops of a computation, with techniques such as software pipelining used to handle the outer loops.

6.3 Vectorizing

Vectorizing is a commonly used optimization for vector supercomputers, executing operations on each vector element in parallel. This technique is highly effective provided that the computation is composed primarily of readily identifiable vector operations (such as dot-product). Most vectorizing compilers generate vector code from a scalar specification by recognizing certain standard looping constructs. However, if the source program lacks the necessary vector-accessing loop structure, vectorizing performs very poorly. For computations that are mostly data-independent, the combination of partial evaluation with static scheduling techniques has the potential to be vastly more effective than vectorization. Whereas a vectorizing compiler will often fail simply because the computation's structure does not lend itself to a vector-oriented representation, the partial-evaluation/static scheduling approach can often succeed by making use of very fine-grained parallelism. On the other hand, for computations that are highly data-dependent, or which have a highly irregular structure that makes unrolling loops infeasible, vectorizing remains an important option.

6.4 Iterative Restructuring

Iterative restructuring represents the manual approach to parallelization. Programmer's write and rewrite their code until the parallelizer is able to automatically recognize and utilize the available parallelism. There are many utilities for doing this, some of which are discussed in [15]. This approach is not flexible in that whenever one aspect of the computation is changed, one must ensure that parallelism in the changed computation is fully expressed by the loop and data-reference structure of the program.

6.5 Static Scheduling

Static scheduling of the fine-grained parallelism embedded in large basic blocks has also also been investigated for use on the *Oscar* architecture at Waseda University in Japan.[6]. The Oscar compiler uses a technique called *task fusion* that is similar in spirit to the grain size adjustment technique used on the *Supercomputer Toolkit.* However, the Oscar compiler lacks a partial-evaluation phase, leaving it to the programmer to manually generate large basic blocks. Although the manual creation of huge basic blocks (or of automated program generators) may be practical for computations such as an FFT that have a very regular structure, it is not a reasonable alternative for more complex programs that require abstraction and complex data structure representations. For example, imagine writing out the 11,000 floating-point operations for the Stormer integration of the Solar system and then suddenly realizing that you need to change to a different integration method. The manual coder would grimace, whereas a programmer writing code for a compiler that uses partial evaluation would simply alter a high-level procedure call.

7 Conclusions

Partial evaluation has an important role to play in the parallel compilation process, especially for largely data-independent programs such as those associated with numerically-oriented scientific computations. Our approach of adjusting the grain size of the computation to match the architecture was possible only because of partial evaluation: If we had taken the more conventional approach of using the structure of the program to detect parallelism, we would then be stuck with the grain size provided us by the programmer. By breaking down the program structure to its finest level, and then imposing our own program structure (regions) based on locality of reference, we have the freedom to choose the grain size to match the architecture. The coupling of partial evaluation with static scheduling techniques in the *Supercomputer Toolkit* compiler also eliminates the need to write programs in an obscure style that makes parallelism more apparent.

Acknowledgements

Guillermo Rozas was a major contributor to the design of the instruction-scheduling techniques we describe in this paper. We would also like to thank Gerald Sussman and Jack Wisdom for the celestial integrators.

This work is a part of the *Supercomputer Toolkit* project, a joint effort between M.I.T. and Hewlett-Packard corporation.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology and at Hewlett-Packard corporation. Support for the M.I.T. laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-4097 and by the National Science Foundation under grant number MIP-9001651. Andrew Berlin's work was supported in part by an IBM Graduate Fellowship in Computer Science.

References

[1] R. Surati, "A Parallelizing Compiler Based on Partial Evaluation", MIT Artificial Intelligence Laboratory Technical Report TR-1377, July 1992

[2] A. Berlin, "A compilation strategy for numerical programs based on partial evaluation," MIT Artificial Intelligence Laboratory Technical Report TR-1144, Cambridge, MA., July 1989.

[3] A. Berlin and D. Weise, "Compiling Scientific Code using Partial Evaluation," *IEEE Computer* December 1990.

[4] A. Berlin, "Partial Evaluation Applied to Numerical Computation," *Proc. 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990.

[5] H. Abelson, A. Berlin, J. Katzenelson, W. McAllister, G. Rozas, G.J. Sussman, and J. Wisdom "The Supercomputer Toolkit: A general framework for special-purpose computing", *International Journal of High-Speed Electronics*, vol. 3, no. 3, 1992, pp. 337-361.

[6] H. Kasahara, H. Honda, and S. Narita "Parallel Processing of Near Fine Grain Tasks Using Static Scheduling on OSCAR", *Supercomputing 90*, pp 856-864, 1990

[7] B. Kruatrachue and T. Lewis, "Grain Size Determination for Parallel Processing", *IEEE Software*, Volume 5, No 1, January 1988

[8] B. Shirazi, M. Wang, and G. Pathak, "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling.", *Journal of Parallel and Distributed Computing*, Volume 10, Number 3, Nov 1990.

[9] E. Ruf and D. Weise, "Avoiding Redundant Specialization During Partial Evaluation" In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluationand Semantics-Based Program Manipulation,* New Haven, CN. June 1991.

[10] E. Ruf and D. Weise, "Opportunities for Online Partial Evaluation", Technical Report CSL-TR-92-516, Computer Systems Laboratory, Stanford University, Stanford, CA. 1992.

[11] N. D. Jones, C. K. Gomard and P. Sestoft, *Partial Evaluation and Automatic Program Generations* Prentice Hall, 1993

[12] M. Lam, "A Systolic Array Optimizing Compiler." Carnegie Mellon Computer Science Department Technical Report CMU-CS-87-187., May, 1987.

[13] J. Ellis, *Bulldog: A Compiler for VLIW Architectures,* MIT Press, Cambridge, MA, 1986.

[14] J.A. Fisher, "Trace scheduling: A Technique for Global Microcode Compaction." *IEEE Transactions on Computers*, Number 7, pp.478-490. 1981.

[15] G. Cybenko, J. Bruner, S. Ho, "Parallel Computing and the Perfect Benchmarks." Center for Supercomputing Research and Development Report 1191., November 1991

[16] J. Applegate, M. Douglas, Y. Gürsel, P. Hunter, C. Seitz, G.J. Sussman, "A Digital Orrery," *IEEE Trans. on Computers*, Sept. 1985.

[17] J. Miller, "Multischeme: A Parallel Processing System Based on MIT Scheme". MIT Laboratory For Computer Science technical report no. TR-402. September, 1987.

[18] G. Sussman and J. Wisdom, "Chaotic Evolution of the Solar System",*Science*, Volume 257, July 1992.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Effects of Loop Fusion and Statement Migration on the Speedup of Vector Multiprocessors

Mayez Al-Mouhamed[*] and Lubomir Bic[**]

[*]Computer Engineering Department, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia

[**]Department of Information and Computer Science, University of California Irvine, Irvine, CA 92717

abstract: Vector multiprocessors rely on both spatial and temporal parallelism for achieving significant speedup. For singly nested loops, we study the effect on the speedup of: 1) loop fusion and, 2) increasing the granule-size of parallel-vector loops using extracted statements from scalar loops. The proposed optimizations migrate vector statements from one loop to another, create new loops, and reduce others. Loops and statements that belong to strongly connected data paths are vertically fused, whenever possible, in order to promote chaining and cache/register reuse. To reduce loop synchronization, horizontal fusion is also used for independent loops having compatible dependence types. Finally, vector operations are scheduled based on knowledge of the timing of arithmetic pipelines, load/store operations, and management of the available resource.

Testing is carried out using synthetic Fortran programs on the Convex $C240$ vector multiprocessor. The proposed loop fusion improves the speedup by 18% to 43% over the C240 commercial optimizing compiler. Chaining-oriented scheduling and allocation yields 9% to 15% improvement over the highest optimization option of the C240 compiler.

1 Introduction

Research of the last decade has generated impressive improvements [13] in the design of parallel vector-processors (VPs), due primarily to decreasing cycle time, the use of faster pipelined memories, and the increasing number of VPs.

Restructuring compilers such as Parafrase [12], PFC [1], UFTN [6] or, V-Pascal [14] perform data dependence analysis of loops in order to classify them depending on their inherent parallelism. Scalar loops (SL) are the least reduceable because of their tight recurrences. Techniques for extracting parallelism out of SL loops have been developed based on recurrence analysis [8]. For example, *Cycle Shrinking* [13] and *Graph Traverse Scheduling* [2] split the loop iteration range into independent or synchronized partitions which can run in parallel.

Loop fusion has been considered [5] as an optimization method for compiling programs targeted to distributed-memory systems. Loop fusion simplifies [10, 11] data partitioning and allows increasing memory reuse. Loop fusion, however, has not been utilized heavily with vector multiprocessors.

Vectorizing compilers support operators to extract parallelism from loops that are primarily classified as scalar or having limited parallelism. Unfortunately, the extracted parallelism is distributed only out of the original loop. The work presented in this paper is based on the use of known techniques for extracting parallelism but with the objective of reducing the granule size of some scalar loops, fusing loops, forming new loops, or migrating statements from one loop to another whenever possible. With this approach, vector scheduling is proposed in order to exploit the locality of data producers and consumers that results from fusion. Scheduling vector operations is based on the use of an accurate model and timing of the underlying vector processor. The benefits are an increase of the granule size of parallel vector loops at the expense of scalar loops, increasing reuse of cache memory and vector registers, improved chaining, and reduced loop synchronization. The approach has been tested on the Convex C240 mainframe.

This paper is organized as follows. Section 2 presents our approach to loop distribution and fusion. Loop scheduling for the Convex $C240$ is developed in Section 3. Resolving conflicts due to register and data path allocations is presented in Section 4. Section 5 presents the evaluation of this work and Section 6 concludes about this work.

2 Loop Distribution and Fusion

Our objective is to investigate the benefit of fusing loops and statements following the extraction of parallelizable computations out of loops that are primarily classified as scalar loops. We do not search to extract all the parallelism out of such loops. There are many approaches to exploit most of the inherent parallelism in loops [12, 13, 15]. The effectiveness of the gained speedup is necessarily dependent on the reduction factor, the granule size of the loop, and the amount of needed synchronization.

In the following we present an algorithm called *Distribute* for reducing the granule size of loops that are primarily classified as scalar loops. Algorithm Distribute classifies loops into a number of categories depending on the amount of available parallelism and synchronization needed to generate correct results. The algorithm for distributing loops can be summarized by the following steps:

1. A loop L is classified as *loop-independent-dependency* (LID) if L does not carry dependency from one iteration to another but dependency may occur across the references of the same iteration. This is true when every pair of references r_1 and r_2 of L satisfy $lcd(r_1, r_2) = 0$, where $lcd(r_1, r_2)$ is the *loop-carried-dependency test.* The test $lcd(r_1, r_2)$ yields true value only when r_1 is referenced in some iteration and assigned in another iteration. An L_{lid} loop can be strip-mined to generate a *parallel vector loop* (PVL). If this step succeeds in classifying L as an L_{lid}, then processing of L terminates and processing of the next loop L' is started.

2. Loop L could be partitioned into a number of L_{lid} loops and one reduced L_{lcd} loop if some expressions of L are only involved in LID dependence which enables distributing each of these expressions out of the original LCD loop. In other terms, there is no reference $r_1 \in L_{lid}$ for which there exists another reference $r_2 \in L$ such that $lcd(r_1, r_2) = 1$, where both r_1 and r_2 reference the same array. Each synthetized L_{lid} loop will be formed by one expression that is free of LCD dependencies. Based on data producer-consumer relationships, any resulting L_{lid} loop should either be predecessor or successor with respect to the remaining L_{lcd} loop in the corresponding dataflow graph.

3. *Partial vectorization* (PV) is attempted in order to reduce the granule size of the current L_{lcd} loop. PV consists of distributing parallelizable and vectorizable statements out of LCD vector expressions. The distributed statements should not be involved in LCD dependence with the remaining expressions of the same loop. Temporary arrays are used to store the results of the distributed statements that become predecessors to their original LCD expressions in the dataflow graph. PV may generates an arbitrary number of L_{lid} loops that will be classified as PVL.

4. A reduction is detected whenever two references r_1 and r_2 of one expression reference a scalar or array with $lcd(r_1, r_2) = 0$ and r_1 is used prior to storing a computed value into r_2. A *reduction* operation can be vectorized by using dedicated vector accumulator, vector compress and merge, thus, leading to a *parallel vector loops with dedicated code* (*PVLD*).

5. The remaining L_{lcd} could be partitioned into a backward-LCD (B-LCD) and a forward-LCD (F-LCD). A F-LCD exists when one iteration references a value that will be assigned in a later iteration. This can be determined by using a write-after-read ($war(r_1, r_2) = 1$) test which can be combined with $lcd(r_1, r_2) = 1$ to indicate the presence of B-LCD dependency between r_1 and r_2. An F-LCD loop does not prevent vectorization because a write-after-read dependence proceeds in the correct order when executed on a vector pipeline. Because some iteration boundaries can concurrently proceeds across the processors, F-LCD loops cannot be safely parallelized without adding synchronization to guarantee correct results. A distributed F-LCD loop is labelled as *parallel vector loops with synchronization* (PVLS). The remainder of the original loop is L_{lcd} with a B-LCD dependence type.

6. Two expressions for which there is a B-LCD dependence can be interchanged if the result generated in one expression is not used in evaluating the second expression and no other dependence is violated. Although, s_1 and s_2 can be involved in B-LCD dependence they can be interchanged without altering the results if s_2 does not reference the computed value of s_1. If s_1 and s_2 are not involved in other dependence, expressions s_1 and s_2 can then be distributed out of the current L_{lcd} loop and two new loops can be created.

7. The remaining unreduceable B-LCD loop generally cannot be vectorized nor parallelized which leads to a *scalar loop* (*SL*) that should be executed in sequence.

Following the above analysis the program is represented by a *data dependence graph* (DDG) in which the nodes represent loops of type SL, PVLD, or PVLS or simply loops of type PVL. The method used for loop fusion is based on *vertical* and *horizontal* fusion:

1. Vertical fusion: statements or loops of the same type and with the same headers which belong to a dependence path are fused whenever possible. Using the previously generated loop labeling, fusing condition for PVL or PVLS loops is examined in order to ensure preserving the loop type of the fused loop. Fusion is not performed if the fused loop could contain: 1) F-LCD (prevent parallelization), or 2) B-LCD (fusion preventing).

2. Horizontal fusion: data independent loops or statements having the same types and loop counts are also fused in order to reduce synchronization and loop overhead.

```
do  i=1,n
      s1: a(i)=a(i-1)+b(i)*c(i)
      s2: e(i)=c(i)-b(i)
      s3: sum=sum+a(i)*sqrt(e(i))
enddo
```

Figure 1: Example of a loop carrying a B-LCD

```
do  i=1,n                     (Parallel-vector loop)
      s2: e(i)=c(i)-b(i)
      s3'': t2(i)=sqrt(e(i))
      s1': t1(i)= b(i)*c(i)
enddo
do  i=1,n                     (Scalar loop)
      s1: a(i)=a(i-1)+t1(i)
enddo
do  i=1,n                     (Parallel-vector loop)
      s3': t3(i)= a(i) * t2(i)
enddo
do  i=1,n                     (Add-reduce loop)
      s3: sum=sum+t3(i)
enddo
```

Figure 2: Output program following Distribution/Fusion

Analysis of conditions that prevent loop fusion can be found in [15]. Loop boundaries in the output program are mainly those corresponding to loops with different types, different loop counts, or same type but with fusion preventing condition.

We examine the program shown in Figure 1 as an example of distributing and fusing loops. The loop has an LCD with respect to references $a(i)$ and $a(i-1)$ which indicates that Step 1 of *Distribute* fails. In Step 2, $s2$ is distributed out of the original loop. In Step 3, PV distributes statements $s1' : t1(i) = b(i) * c(i)$, $s3'' : t2(i) = sqrt(e(i))$, and $s3' : t3(i) = a(i) * t2(i)$ out of the the original LCD loop, where $s1'$, $s3'$, and $s3''$ are identifiers for the three new statements. Statements $s1'$, $s3'$, and $s3''$ are labeled as PVL loops because they are free of LCD dependencies. The previous statements $s1$ and $s3$ are then reduced to $s1 : a(i) = a(i-1) + t1(i)$ and $s3 : sum = sum + t3(i)$ as a result of the previous partial vectorization. Step 4 finds the reduction that is present in $s3$ and creates a new loop for $s3$ that is labeled PVLD. The remaining LCD loop is reduced to $s1$. Step 5 indicates that $s1$ is a B-LCD and there is no F-LCD. Step 6 fails because the B-LCD loop is formed by one single expression $s1$ that is labeled as SL loop.

The resulting data dependence graph has four levels: $l1 = \{s1', s2\}$, $l2 = \{s1, s3''\}$, $l3 = \{s3'\}$, and $l4 = \{s3\}$. The dependence edges are: $(s1' \rightarrow s1)$, $(s2 \rightarrow s3'')$, $(s1, s3'' \rightarrow s3')$, and $(s3' \rightarrow s3)$.

Vertical fusion leads to fuse expressions $s2$ and $s3''$. As $s1$ is SL and $s2$ is LID, horizontal fusion inserts then term $s1'$ within loop ($s2$,$s3''$) that becomes ($s1'$,$s2$,$s3''$). The type of $s3'$ is different from that of $s1$ or $s3$. The above three loops remain separate as shown in Figure 2.

3 Loop Scheduling

The traditional approaches [7, 9] are based on the use of *reservation tables* that result from gross estimating the functional unit (*Fus*) times as one or more units of times and scheduling the vector operations. Resolving problems with respect to memory load and store operations, resource availability, and resource allocation is done in a separate step. The result is that isolating the above constraints from the first step leads schedule makespan to be excessively increased by adding delays for resolving the various register and data-path constraints in the final step.

Our approach consists of *early incorporation of all the available constraints* by scheduling the load/store and arithmetic operations over the *Fus* based on accurate timing of the VP and the management of the available resources during scheduling. This results in a conflict-free schedule with respect to *Fus*, memory utilization, and resource availability. The schedule generated is used to resolve the problem of register and data-path allocation. This consists of resolving conflicting allocation of registers and data-paths by minimizing the additional delays to be incorporated in the original schedule.

The vector processor unit of C240 [7, 4] has the following vector pipelines (*Fus*): 1) concurrent vector load (M_{out}) and store (M_{in}), 2) add and logical (Fu_1), 3) multiply and divide (Fu_1), and 4) merge/compress and conditional pipes. Any of the four register bank outputs can route vector data to any of three functional pipelines inputs. The functional unit (Fu) output and M_{out} can be routed to any of the register bank inputs. Each bank has two vector registers but only one bank input.

In the following we investigate evaluation of the *earliest-starting-time* ($est(n)$) of vector operation n. We start by defining the notation used prior to finding the *est* time for each possible case. Variable *vs* denotes a vector or a scalar. $vs.n$ denotes a *vs* that is input operand for vector operation n. $n.vs$ denotes a vector/scalar that is produced by n. $Fu(n)$ denotes the functional unit on which n is to run. Denote by $s(n)$, $t(n)$, $t_l(vs)$ the starting time of n, finishing time of n, and the time to load *vs* from memory into some register, respectively. $t(M)$ and $t(Fu)$ denote the earliest time the memory (M_{in} or M_{out}) and functional unit Fu are free, where M_{out} (loading) and M_{in} (storing) are memories used to establish two independent data-paths (C240) with the vector-processor.

Evaluation of the earliest-starting time ($est(n)$) is based on the previous status of the *Fus*, M_{in} and M_{out}, and the number of registers and data-paths used. Evaluation of the $est(n)$ for the C240 requires analysis of the following three cases.

For Case 1, n requires loading of two vectors $vs_1.n$ and $vs_2.n$. Denote by $t_L(vs_1, vs_2) = t(M_{out}) + t_l(vs_1) + t_l(vs_2)$ the time at which both operands $vs_1.n$ and $vs_2.n$ will be loaded from the memory into some registers. As there is no possible chaining, $est(n)$ is defined by $est(n) = \max\{t_L(vs_1, vs_2), t(Fu(n))\}$.

Case 2, n requires loading of one vector $vs.n$ while the other operand is generated by a predecessor n'. If no chaining is possible ($Fu(n) = Fu(n')$), then $est(n)$ depends on the later of the loading time $t_L(vs)$ and the time $t(Fu(n'))$ the functional unit $Fu(n')$ is free, i.e., $est(n) = \max\{t_L(vs), t(Fu(n'))\}$.

In the other case ($Fu(n) \neq Fu(n')$), chaining can be done if at the chaining time ($t_c(n) = s(n') + \delta(n')$) the loading of $vs.n$ is complete and $Fu(n)$ is free, where $\delta(n')$ is the delay on of n due to its chaining with operation n'. If the previous condition is not satisfied, then running n may take place following the completion of n'. To summarize, the $est(n)$ will be defined by:

$$est(n) = \begin{cases} t_c(n) & \text{if } t_c(n) \geq \max\{t_L(vs), t(Fu(n))\} \\ \max\{t_L(vs), t(n'), t(Fu(n))\} & \text{otherwise (no chaining)} \end{cases}$$

For Case 3, both operands of n are generated by predecessors n' and n'', respectively. If there is no chaining $(Fu(n) = Fu(n') = Fu(n''))$, then $est(n) = t(Fu(n))$ because no loading is needed.

There is potential chaining on one unit when $(Fu(n) \neq Fu(n'))$ but $(Fu(n) = Fu(n''))$. Chaining is possible if at the chaining time $t_c(n) = s(n') + \delta(n')$ the computation n'' is complete and $Fu(n)$ is free, where $\delta(n')$ is the time to store the first result of n'. In other words, $est(n)$ is defined by:

$$est(n) = \begin{cases} t_c(n) & \text{if } t_c(n) \geq \max\{t(n''), t(Fu(n))\} \\ \max\{t(Fu(n)), t(Fu(n''))\} & \text{otherwise (no chaining)} \end{cases}$$

There is potential chaining on two units when $Fu(n) \neq Fu(n')$ and $Fu(n) \neq Fu(n'')$. We restrict ourselves to the case of the $C240$ that has two arithmetic units. Therefore, operations n' and n'' are sequentially executed and the chaining time of n is $t_c(n) = \max\{t'_c(n), t''_c(n)\}$, where $t'_c(n) = s(n') + \delta(n')$ and $t''_c(n) = s(n'') + \delta(n'')$. The chaining is possible if $Fu(n)$ is free at the chaining time $t_c(n)$, otherwise n is to start following the completion of both predecessors n' and n''. To summarize this case, we have:

$$est(n) = \begin{cases} t_c(n) & \text{if } t_c(n) \geq t(Fu(n)) \\ \max\{t(Fu(n)), t(Fu(n')), t(Fu(n''))\} & \text{otherwise (no chaining)} \end{cases}$$

Finally, we evaluate the earliest time $(t_{rc}(n))$ at which the resource (Fu, M_{out}, and bank inputs and outputs) needed for n becomes available. The register and data-path availability are examined at time point $est(n)$ that has been evaluated previously so that $t_{rc}(n) \geq est(n)$. If the needed resource is available, then $t_{rc}(n) = est(n)$. Otherwise, $t_{rc}(n)$ will be the earliest time the needed resource will be released. The final earliest-starting time of n will be used by the scheduling algorithm.

Using the above evaluation of the earliest-starting-time, the proposed vector-scheduling is based on minimizing the Fu idle times in an attempt to maximize the efficiency and to promote potential vector-chaining. Each loop separately scheduled. Vector load is not considered as a separate operation but included within each of its arithmetic vector operation. At any given time, the operands of ready-to-run operations are available in memory or in some vector registers. The major steps of the *Vector-Scheduling* algorithm is the following:

1. All operations of the loop are stored into set A except the set of ready-to-run operations that are stored into set B

2. Repeat the following steps until B is empty

   (a) Select the operation $n \in B$ that can start at the earliest time $(est(n))$ as a heuristic to maximize the efficiency of the Fus, allocate n, allocate one register to the output of n, and update the timing of all the resources used by n

   (b) Load input operands (if any) for n, allocates one register, and updates the time M_{out} will be free

   (c) Check whether a successor operation of n becomes ready to run, remove it from A and insert it into B

The time complexity of the vector-scheduling algorithm is $O(pm^2)$, where p is the number of resources and m is the number of vector operations.

4 Minimizing Register and Data-Path Conflicts

The previous scheduling guarantees that: 1) different loads (M_{out}) or stores (M_{in}) are properly serialized, 2) allocation of vector operations to *Fus* do not conflict and, 3) resource availability is guaranteed. In the following we analyse three types of conflicts that might result from the use of the previous schedule.

Overlapped lifetime of vectors may lead to conflicting utilization if at least two vectors are allocated to the same register or to different registers but within the same register bank. Denote by $\omega_{sp}(v, v')$ the time cost, or delay over the finish time of the previous schedule, of spilling either v or v' whenever their lifetimes overlap. The cost $\omega_{sp}(v, v')$ is the time to store and load one vector data.

Overlapped bank output occurs when two vectors v and v' are allocated to registers of the same bank and their loading into the corresponding *Fus* overlap with respect to time. Denote by $\omega_{bout}(v, v')$ the cost associated with the least delay incurred to the schedule as a result of serializing the use of the bank output. Therefore, $\omega_{out}(v, v') = \min\{t(n') - s(n), t(n) - s(n')\}$, where n and n' are the operations that consume v and v', respectively, and $s(n)$ and $t(n)$ are the starting and ending times of n.

Overlapped bank input occurs when two vectors v and v' are allocated to registers of the same bank and their storing into their registers overlap with respect to time. The associated cost ω_{bin} is the least delay incurred by the schedule when serializing the use of the bank input.

Based on the above time delays, we heuristically define the global weight $\omega(v, v')$ of edge (v, v') as the maximum delay caused by the interfering operations of vectors v and v'. Each vector v is further assigned a weight $\omega(v)$ that is the sum of all the delays caused to the schedule:

$$\omega(v) = \begin{cases} o & \text{if no lifetime overlap} \\ \Sigma_{v'} \max\{\omega_{sp}(v, v') + \omega_{bin}(v, v') + \omega_{bout}(v, v')\} & \text{otherwise} \end{cases}$$

In the following we use *Weighted Graph Coloring* for allocating banks and registers to the vectors of each loop. The vectors used in each loop are associated an undirected graph in which a node v has the weight $\omega(v)$ and an edge (v, v') has the weight $\omega(v, v')$. The graph is formed by a collection of non-connected sub-graphs. The used coloring algorithm is a variant of [3]. The algorithm uses two heaps A and B:

1. Initially, A contains all nodes except the one with the highest weight which is placed in B,

2. Repeat the following steps until A is empty

   (a) Repeat the following step until B is empty

      i. Remove a node v from B (highest weight), color v, and add to B all immediate neighbors of v after removing them from A

   (b) Remove from A the node (first node of a new component) with the highest weight and insert it into B

The time complexity of this algorithm is $O(cm^2)$, where c is the number of components and m is the number of vectors within a component.

Different Programs and Different Allocations

Speedup	L(1)	L(2)	L(3)	L(max)
$S_{C240/nopt}$	11	9.75	8.72	8.6
$S_{opt/nopt}$	15.73	12.87	10.55	10.15
$S_{opt/C240}$	1.43	1.32	1.21	1.18

Table 1: Speedup of ($C240$) and this Approach (opt)

5 Performance Evaluation

Our objective is to develop a compiler optimization to assists non-expert programmers. We consider synthetic Fortran programs. Each is formed by a total of 30 arithmetic vector-expressions. Each expression may include up to four vector-operations. To generate these programs, the dependence edges between the expressions were randomly generated in order to specify the vector-operands for each expression including possible vector-loads. The maximum number of expressions that belong to the same level in the data dependence graph (DDG) were at most four and the number of levels was at least five. An LCD was present in 20% of the expressions and all loop counts were set to 1000. We use $L(k)$ to denote that each loop is allowed to incorporate at most k expressions.

Let's $S_{C240/nopt}$ be the speedup of the highest optimization level of the Convex $C240$ over the No-Optimization or scalar execution. We also use the speedup $S_{opt/nopt}$, where *opt* refers to the proposed approach. Finally, we use the Speedup $S_{opt/C240}$ to compare our approach to that of the $C240$ highest optimization option. All the studied cases use one processor only. Table 1 shows the results of averaging the running of 10 programs for each value of $L(k)$.

The first entry in Table 1 shows the measured speedup of the $C240$ optimizer over the scalar processor ($-no$) when only one VP is used. The highest speedup is 11 which is measured when the program is formed by loops that contain only one vector expression. This speedup decreases with increasing the granule size of the loops ($L(1)$, $L(2)$, etc.). The reason is that the vector processor of the $C240$ has hardware support to implement the loop overhead that is much faster than the software approach used by the scalar processor.

With loops having few statements ($L(1)$, $L(2)$, and $L(3)$), the newly loaded vector data in the beginning of each loop could have been made available in vector registers during the running of previous loops. The overhead of multiple vector load and store is avoided in our approach through loop fusion. The effect of loop fusion allows improving the speedup as shown in $S_{opt/nopt}$ and $S_{opt/c240}$ of table 1. All assignment statements require storing data regardless of loop boundary. The difference is that with loop fusion some of these vector store can occur while the arithmetic pipelines can be active.

The effect of statement migration appears more sharply when the number of expressions within each loop is maximum ($L(max)$). In this case, the opportunity of loop fusion is reduced but there is still some benefit from transferring expressions from one loop to another in order to align vector data producers with vector data consumers as an attempt to promote pipeline chaining. The benefit of statement migration ($L(max)$) allowed to improve the speedup obtained by the $C240$ by 18% as shown by $S_{opt/c240}$.

Same Programs but Different Allocations

Speedup	L(1)	L(2)	L(3)	L(max)
$S_{C240/nopt}$	13.69	11.50	9.68	9.31
$S_{opt/nopt}$	15.73	12.87	10.55	10.15
$S_{opt/C240}$	1.15	1.12	1.09	1.09

Table 2: Effect of the allocation and scheduling of the VP

5.1 Effect of Scheduling and Allocation

The objective of the second experiment is to study the effects of the proposed vector scheduling and resource allocation. For this, the $C240$ optimizer and our scheduler were run using the same restructured program in an attempt to isolate and compare the scheduling and allocation effects which have been done separately.

Table 2 shows the measured speedup for this experiment. The achievement of our scheduling and allocation are still those obtained in the first experiment. However, the $C240$ optimizer improves its speedup ($S_{C240/nopt}$), when using restructured programs, by a 10% to 24% factor compared to that achieved without restructuring the program.

The $C240$ can find more opportunities for chaining in the restructured programs as most producers/consumers are now located within the boundary of each loop. Our approach improves its speedup by a 9% to 15% factor over that of the $C240$ due to our scheduling and allocation.

The $C240$ optimizer uses the technique of reservation table in scheduling vector operations over the available pipelines [7]. The timing of all the operations is rounded so that it can either be one or two macro-cycles. Vector scheduling that is a variant of *list-scheduling* [9] is performed in the first pass. There are many reasons for inserting delays into the reservation table in the allocation pass. In general, the inserted delays negate some of the chaining benefits of the first step. A conservative conclusion is that early incorporation of timing constraints in allocating the vector registers and data-paths yields better result than the approach used in commercial vectorizers such as that of the C240.

6 Conclusion

The objective of this work was to investigate some aspects of global restructuring of synthetic programs based on restructuring of the source program and the use of a chaining-oriented machine dependent allocation. For this, we proposed reducing the granule size of scalar loops by extracting vectorizable and parallelizable statements. Furthermore, these statements can be fused within other loops of the same type and having similar loops count in an attempt to increase the granule size of parallel loops. The other benefit of this operation is that placing data producers closer to their consumers increases the opportunity for chaining their execution, and increasing reuse of data in vector-registers and cache-memory. At a lower level, we proposed a *horizontal scheduling* and allocation method based on locally maximizing the efficiency and incorporation of the vector load/store delays.

Our approach allows improving the highest optimization option of the convex C240 by a speedup factor of 18% to 43%. Chaining-oriented scheduling and allocation gave an improvement of 9% to 15%.

Future extension to this work would be the integration of expert techniques in parallelism extraction for each type of loops. This would greatly help scientists in optimizing large scale programs within the framework of an interactive optimizer.

References

[1] K. Allen, J.R. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9, No 4:491–542, Oct 1987.

[2] E. Ayguade et al. GTS: Parallelization and vectorization of tight recurrences. *Proc. of the ACM conf. on Supercomputing*, pages 531–539, 1989.

[3] G. J. Chaitan. Register allocation and spilling via graph coloring. *ACM SIGPLAN Notices*, 17(2):201–207, 1982.

[4] G. Chastain, M. Gostin and J. Mankocich. The Convex C240 architecture. *IEEE Proc. on Supercomputing*, pages 321–329, 1988.

[5] A. Choudhary and S. Hiranandani. Compiling Fortran 77D and 90D for MIMD distributed-memory machines. *Proc. FMPC-92*, pages 4–11, 1992.

[6] H.B. Coleman. The vectorizing compiler for the UNISYS ISP. *Proc. of the 1987 ICPP*, pages 567–576, Aug 1987.

[7] CONVEX Computer Corporation. The Convex Hardware Manual. *Convex Press, Richardson, Texas*, Feb 1987.

[8] R.G. Cytron. Doacross: Beyond vectorization for multiprocessors. *Proc. of the 1986 ICPP*, pages 836–844, Aug 1986.

[9] W. Eisenbeis, C. Jalby and A. Lichnewasky. Squeezing more CPU performance out of a CRAY-2 by vector block scheduling. *IEEE Proc. on Supercomputing*, pages 237–246, 1988.

[10] k. Kennedy and K. S. McKineley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. *ACM Conference on Programming Language Design and Implementation*, 1989.

[11] k. Kennedy and K. S. McKineley. Optimizing for parallelism and data locality. *Inter. Conf. On Supercomputing, Washington, DC*, 1992.

[12] D.J. Kuck et al. The structure of an advanced vectorizer for pipelined processors. *Tutorial on Supercomputers: Designs and Applications, K. Hwang ed*, pages 163–178, 1984.

[13] C.P. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, London, 1988.

[14] T. Tsuda and Y. Kunieda. V-Pascal: an automatic vectorizing compiler for Pascal with no language extensions. *IEEE Proc. on Supercomputing*, pages 182–198, 1988.

[15] J. Warren. A hierarchical basis for reordering transformation. *Conference Record of the 11th Annual ACM Symp. on Principles of Programming languages*, Jan 1984.

PART VII

LOGIC LANGUAGES

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Practical Static Mode Analyses of Concurrent Logic Languages

E. Tick

Dept. of Computer Science, University of Oregon, Eugene, OR 97403, USA

Abstract: A popular approach to speed up fine-grain concurrent languages is to partition programs into threads. This requires static analysis to determine task dependencies, to avoid placing a cycle within a thread. In the context of concurrent logic programs (CLPs), dependency analysis requires *mode analysis.* Simple argument modes are insufficient because dependencies can be hidden within complex terms. In this paper we review Ueda and Morita's proposed *path analysis method* and present three novel algorithms for realizing the technique. We present empirical measurements of the analysis times of a benchmark suite which indicate that the analysis can be comparable to the compilation time on a simple, non-optimizing compiler. This study presents the first empirical results concerning the practicality of complete mode analysis for CLPs.

Keyword Codes: D.1.3; D.1.6
Keywords: Programming Techniques, Concurrent Programming; Logic Programming

1 Introduction

Concurrent logic programs, such as FGHC [9], as well as other dataflow languages such as Id, are collections of fine-grain concurrent tasks that synchronize implicitly when input data is not available to a procedure invocation. Naive compilation causes too many, too small tasks to be generated. Improving execution performance by static partitioning into threads is a currently developing technique e.g., [6]. In contrast to functional languages, logic languages have an additional problem that data dependencies are implicit, i.e., the input/output characteristics of program variables are not explicitly declared or trivially derivable. Conservative knowledge of dependencies is necessary to avoid placing a cycle within a thread thereby causing erroneous deadlock.

There are numerous methods for automatic derivation of mode information from sequential logic programs, e.g., [2, 4, 5]. Another option is user declarations (e.g., [10]), which we consider either incomplete or too much burden on the programmer. For CLPs, Ueda and Morita [14] proposed a mode analysis scheme based on the representation of procedure paths and their relationships as rooted graphs ("rational trees"). Unification over rational trees combines the mode information obtainable from the various procedures. For example, in a procedure that manipulates a list data stream, we might know that the mode of the car of the list (that is the current message) is the same mode as the cadr (second message), caddr (third message), etc. This potentially infinite set of "paths" is represented as a concise graph. Furthermore, a caller of this procedure may constrain the car to be input mode. By unifying the caller and callee path graphs, modes can be propagated. This analysis is subtlely different than classical abstract interpretation, which could also be used to generate identical information.

Mode information is useful not only for compiler optimization but also for static bug detection. In the latter, the analyzer warns the programmer that variable usage disobeys conventions and is thus likely to be erroneous. In this regard, the analysis constitutes a type restriction on the language, called "moded" flat committed-choice logic programs by Ueda and Morita. These are programs in which a variable has at most a single producer and the mode of each path in a program is constant, rather than a function of the occurrences of the path. This is not regarded as a major drawback, since most non-moded flat committed-choice logic programs can be transformed to moded form.

To motivate the importance of the static analysis, consider an experimental FGHC-to-C compiler built using these techniques [7]. Measurements were made comparing the sequential C code generated by our sequentializer and several other systems. Noteably, handcrafted C programs and FGHC programs compiled by the Monaco research compiler generating parallel code [11]. For a QuickSort program sorting 500 integers on a single Sequent Symmetry processor, Monaco ran in 10.5 seconds, handcrafted C in 1.5 seconds, and our sequentialized code in 2.2 seconds. We chose this example to motivate the point that significant performance improvements over traditional systems (Monaco was the fastest *multiprocessor* implementation of FGHC at the time of the experiment) can be achieved with this technique, and the speeds are getting closer to optimized C. Although more sophisticated partitioning (e.g., based on granularity estimation [6]) is needed to retain multiple threads for parallelism, mode analysis is still required for safety.

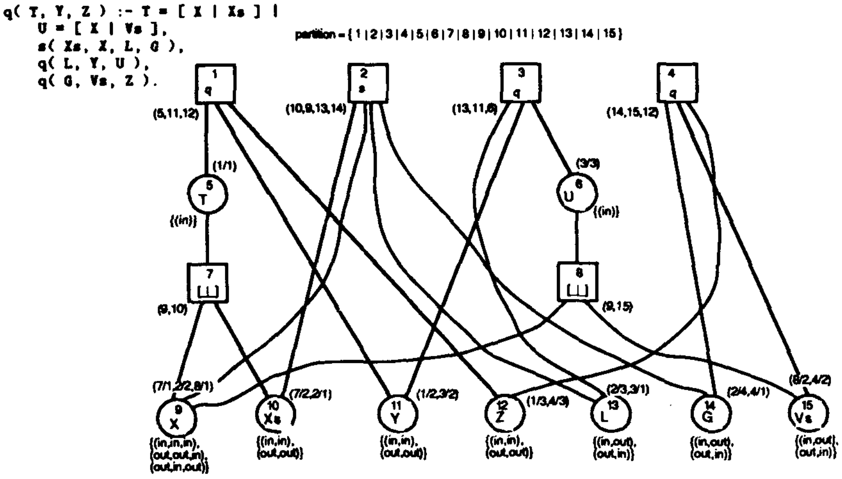
In this paper we discuss three algorithms for implementing Ueda and Morita's technique: the first implementations yet built and evaluated (M. Koshimura also has an MGTP implementation [12]). We present empirical measurements of the analysis times of a benchmark suite which indicate that the analysis is comparable to compilation time on a simple, non-optimizing compiler.

2 Background: Paths and Modes

Ueda and Morita's [14] notion of "path" is adopted as follows: a path p "derives" a subterm s within a term t (written $p(t) \vdash s$) iff for some predicate f and some functors $a, b, \ldots$ the subterm denoted by descending into t along the sequence $\{< f, i >, < a, j >, < b, k >, \ldots\}$ (where $< f, i >$ is the i^{th} argument of the functor f) is s. A path thus corresponds to a descent through the structure of some object being passed as an argument to a function call. f is referred to as the "principal functor" of p. A program is "moded" if the modes of all possible paths in the program are consistent, where each path may have one of two modes: *in* or *out* (for a precise definition, see Ueda and Morita [14]). For example the cadr of the first argument of procedure q has an input mode specified as: $m(\{< q/3, 1 >, < ./2, 2 >, < ./2, 1 >\}) = in$.

All analyses presented in this paper exploit the rules outlined by Ueda and Morita. For the purposes of following this paper, we grossly summarize these as three rules: §1 nonvariables and variables constrained in certain ways in the guard are immediately deduced as *in*; §2 corresponding paths among left and right side of body unification goals have *opposite* modes, and §3 variables have at most one producer thus multiple occurrences have at most one *out* instance.[1]

[1] This is not precise: a variable produced in the body (*out*) can be exported in the head (also *out*).



```
q( T, Y, Z ) :- T = [ X | Xs ] |
    U = [ X | Vs ],
    s( Xs, X, L, G ),
    q( L, Y, U ),
    q( G, Vs, Z ).
```

Figure 1: Initial Graph of Procedure q/3, After Phases I-II

3 Constraint Propagation Algorithm

In the constraint propagation algorithm [13], a graph[2] is constructed representing the entire program.[3] Analysis proceeds by unifying all roots with the same functor and arity. Termination, occurring when no further reduction is possible, is guaranteed because the mode lattice is finite and monotonic. Checking for program "well modedness," the analysis cannot terminate even if all modes are derived, in anticipation of a later contradiction.[4] Thus time complexity is simply a function of the size of subgraphs to be unified [14] which are usually small. A program graph is a directed, multi-rooted, (possibly) cyclic graph composed of two types of nodes. To illustrate the following definitions, Figure 1 presents a portion of the program graph for Quicksort.

A *structure node* (drawn as a square) represents a functor with zero or more exit-ports corresponding to the functor's arity. If the node corresponds to a procedure name (for clause heads and body goals), there are no associated entry-ports (i.e., it is a root). If the node corresponds to a data structure, there is a single entry-port linked to a *variable node* unified with that term. A structure node contains the following information: a unique identifier, functor, and arity.

A *variable node* (drawn as a circle) represents a subset s of (unified) variables in a clause. Intuitively we think of these variables as aliases, and upon initial construction of the graph, s is a singleton (i.e., each unique variable in the clause has its own variable node initially). A node contains $k \geq 1$ entry-ports and $j \geq 0$ exit-ports, upon which directed edges are incident. A unique entry-port corresponds to each clause instance of each variable in s. An exit-port corresponds to a possible unification of the variable(s) to

[2] Throughout the paper we informally refer to rational trees as graphs. Note that our graph grammar is quite different than that of Ueda and Morita.

[3] *It is straightforward to build an incremental analyzer; we do not go into details here.*

[4] A program is "well-moded" if the modes of *all* paths are known, "moded" if the modes of *some* paths are known, and "non-moded" if there is a mode *contradiction*.

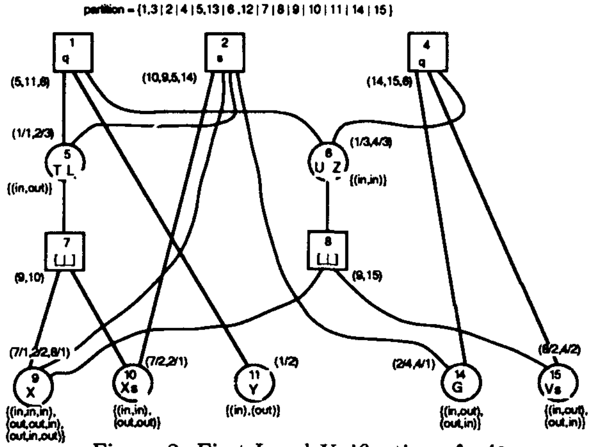


Figure 2: First Local Unification of q/3

a term (exit-ports connect to *structure nodes*).

A variable node contains the following information: a unique identifier and a *mode set m*. An element of m is a vector of length k containing self-consistent modes for the variable *instances* of s. To facilitate the implementation, each entry-port has a *name*: the identifier and exit-port number of its source node. Elements of m are *alternative* mode interpretations of the program. Initially m is computed by Ueda and Morita's rules.[5] Intuitively, graph reduction results in removing elements from m as more constraints are applied by local and global unifications. A fully-reduced graph, for a well-moded program, has a singleton m in each variable node.

Initial graphs, e.g., Figure 1, are multi-rooted directed acyclic graphs. The initial roots correspond to clause head functors, body goal functors, and body unification operators. In addition to the program graph, a *partitioned node set* is kept. Initially, each node is a singleton member of its own partition (disjoint set).

The mode analysis consists of three phases: I) creating a normalized form and initial graph; II) removing unification operators from the graph, and III) reducing the graph to a minimal form. Phases I and II (shared by the process network implementation) convert a flat committed-choice program into a normalized graph with roots named only by clause heads and user-defined body goals. Precise details of this transformation can be found in [13]. Phase III is described next.

Throughout this brief discussion of unification, consider the example shown in Figure 1 (initial) and Figure 2 (after unification of roots 1 and 3). See [13] for the formal graph unification algorithm. Unification is invoked as *unify(a,b)* of two nodes a and b (necessarily root structure nodes). The result is either failure, or success and a new graph (including the node partitioning) that represents the most general unification (mgu) of the two operands. Implied data structures used by the algorithm include the graph, the

[5] The size of m increases with the complexity of the rules, e.g., rule §3 (Section 2) can produce several vectors. By explicitly enumerating initial modes, we simplify the analysis by obviating the need to actively apply complex constraints implied by rule §3 throughout reduction of the graph.

disjoint sets (i.e., node partitioning), and a *mark table* associated with pairs of nodes.

Structure and variable node unifications follow recursive descents. Initially all marks are cleared. Circular structures that represent infinite paths are handled properly by *marking* node pairs at first visit. If a given node pair has been previously marked, revisiting them immediately succeeds. Note that we mark *pairs* instead of individual nodes to handle the case of unifying cyclic terms of unequal periodicity.

Two important operations for the disjoint sets data structure are *union(x,y)* and *find_set(x)*. Function *union(x,y)* unites two disjoint sets, where x belongs to the first disjoint set and y belongs to the second disjoint set. Procedure *union* returns the *canonical name* of the partition, i.e., the least identifier of the nodes. This facilitates reusing graph nodes while rebuilding the graph. Function *find_set(x)* returns the canonical name of the disjoint set containing x.

The major complexity in the algorithm is variable node unification where the modes of two argument nodes must be merged. First, mode vectors that are contradictory are discarded. If all mode vectors are contradictory then a mode error has occurred and unification fails. Otherwise redundant modes are removed and the two mode vectors are concatenated. Next we create the entry-port identifiers associated with the new mode vector. Lastly, children of the argument nodes that share equal functor/arity must be recursively unified. The exit-port identifiers consist of a single exit-port for each pair of children unified, included with exit-ports for all children for which unification does not take place. Intuitively, a variable node forms or-branches with its children, whereas a structure node forms and-branches with its children. In other words, the least-upper-bound (lub) of the abstract unification semantics at a variable node is a union of the structures that potentially concretely unify with the variable node.

4 Process Network Analyzer

The previous constraint propagation algorithm was alternatively implemented by a process network wherein each node of the graph was an active, concurrent process. Nodes communicated by message passing over streams to accomplish reduction. The motivations for moving the graph from a static data structure to an active process network are: 1) concurrency is increased because updating the graph no longer bottlenecks the computation; 2) unification of graph nodes corresponds to merging node processes, thus resource requirements made by the analyzer decrease as execution proceeds, and 3) an active process network is an elegant paradigm for this problem.

Translating the unification algorithm requires the specification of how recursive unification can proceed via message passing, how the distributed unification can terminate (both successfully and by failure), and how the final mode information can be read from the reduced graph. Do to insufficient space, we briefly discuss only the distributed unification algorithm (see [12] for additional details).

A `node` process manages either a variable or structure graph node. The `node` process contains state holding a unique integer identifier, a symbol (functor/arity for structure nodes and the atom `'$VAR'` for variable nodes), mode information, and a flag indicating if the node is from a clause head. Mode information consists of a set of mode vectors and a vector of entry ports, as described in Section 3. In addition, a `node` has an input stream, a list of output streams to children, and a global termination flag.

A `node` process acts on the following two messages (in addition to others). Receipt of message `unify(+Id,+S0,-S1,+Parents,+Ans,+Done)` indicates that this node is requested to initiate a unification with node `Id` on input stream `S0`. `Parents` are the two parent nodes who made this unification request. The results of the unification are `S1` which is the tail of the stream to node `Id` and `Ans`, a short-circuit chain[6] for unification termination. `Done` is the short-circuit chain for message termination. Receipt of message `who(-Info,-In,-Out,+Done)` indicates that this node is to be unified with another node, and therefore this node is to be terminated. Before termination, the state of the node is passed back to the initiator node via back-messages: `Info`, `In`, and `Out`. `Done` is the short-circuit chain for message termination.

The implementation shares phases I and II with the previous algorithm, and then spawns a node process network from the static graph definition. Similar to the static graph analyzer, the root list is grouped into pairs which are unified, then these resulting trees are unified and so on, forming a logarithmic tree of unifications.

A `node` that receives a `unify/6` message is the "active" member of a reduction. It sends a `who/4` message to the "passive" member, who returns all its state information on a back message and terminates itself. For structure–structure unification, the node symbols are compared and if matched, the active node sends `unify/6` messages to one member of each pair of children. Otherwise failure occurs.

For variable–variable unification, first the mode sets must be merged. If the merge is successful, then only children with matching functors are unified, by sending `unify/6` messages. Non-matching children are simply appended to the output stream list of the active node. If the mode set merge is a failure, then the unification fails.

The primary difference between the previous static (Section 3) and active (Section 4) graph implementations is that the latter is fully concurrent. The static graph is sequentialized by necessity to update the graph consistently. One fix would be to partition the graph into independent subgraphs (finding the strongly-connected components of the call-graph), allowing concurrent reduction. The active graph analyzer was more difficult to build than the static analyzer because concurrent process networks are notoriously hard to debug. However, compared to other distributed algorithms, debugging was not overly burdensome because our abstract unifications monotonically approach the final state.

From profiling information we determined that the active graph analyzer spends most of its time checking for self-unification of a node (necessary for circular unification) and (to a lesser degree) manipulating mode vectors. To check for self-unification, we instituted a naming scheme wherein the identifiers of two nodes to be unified are concatenated to form the identifier of the new node. Thus node identifiers grow in size during reductions, and although we use difference lists to concatenate cheaply, the cost of checking membership within an identifier list grows. An alternative would be to allow both nodes to live (currently we terminate one of them to save space), and update the state in each to indicate the current minimum identifier of the alias set. We have not yet experimented with this option (it is very similar to method used in static graph implementation).

Mode vector manipulation requires finding the indices (within the vectors) of the mode elements being compared, and concatenation of the two vectors (less the duplicate mode element which is removed). Time is spent about equally between these main functions. Quickly finding indices requires a more sophisticated data structure than the current list. Quick concatenation requires either difference lists or bit vectors. Both are complicated by the removal of duplicate elements. In fact, the static graph implementation elected to forgo duplicate removal and used difference lists for mode vectors. This contributes to the increased space requirement for the static graph analysis (Section 6). The active graph implementation uses standard lists with removal. We need further experimentation to determine the best solution.

[6] See [9] for discussion of CLP programming techniques such as short circuits and back messages.

The space complexities of the active graph analyzer lie in spawning a process for each graph node. This working set churns through memory more quickly than the static graph implementation (which can exploit local memory reuse in PDSS to keep data copying low). Currently we do not constrain the number of processes, but this could be accomplished in the manner opposite to parallelizing the static graph analyzer: first finding groups of strongly-connected components of the program's call graph, and then analyzing only one group at a time. For example, a short-circuit chain could be used to force synchronization between one group and the next. In a multiprocessor system, explicit load distribution of the groups would be needed.

5 Finite Domain Analysis

To avoid circular unification altogether and much of the overheads of maintaining the graph, either statically or actively, a radically different algorithm was developed [7]. The first stage of this alternative algorithm generates a finite set of paths whose modes are to be considered. Only "interesting" paths are generated in the first stage of our algorithm: effectively those paths locally derived from the syntactic structure of the procedures. There are three classes of interesting paths. The first class consists of paths that directly derive a named variable in the head, guard, or body of some clause. All such paths can be generated by a simple sequential scan of all heads, guards, and body goals of the program.

The second class consists of paths which derive a variable v in some clause, where a proper path through the opposite side of a unification with v derives a variable v'. More formally, consider a unification operator $v = t$ where v is a variable and t is some term other than a variable or ground term. Let v' be a variable appearing in t at path q, i.e., $q(t) \vdash v'$. Then if p is a path deriving v (by which condition p is also interesting), then the concatenated path $p \cdot q$ is also an interesting path. All paths in this second class may be generated by repeated sequential scanning of all unification goals until no new interesting paths are discovered. The necessity for repeated scans is illustrated by such clauses as "$a(X, Z)$:- $Y = c(X)$, $Z = b(Y)$." where the interesting path $\{< a/2, 2>, < b/1, 1>, < c/1, 1>\}$ given by the first unification body goal will not be generated until the interesting path $\{< a/2, 2>, < b/1, 1>\}$ in the second unification body goal is generated. Such repeated scans should occur infrequently in practice. In any case not more than a few scans are necessary — no greater number than the syntactic nesting depth of expressions containing unification operators.

The third class of interesting paths is generated by noting that if a path starting on the right-hand side of a unification body goal (i.e., a path of the form $\{<=/2, 2>\} \cdot s$) is interesting, then so is the corresponding path starting on the left-hand side of that unification (i.e., $\{<=/2, 1>\} \cdot s$).

In general, all interesting paths of a program are generated in a few sequential passes, e.g., the 39 interesting paths of Quicksort are generated in two passes. The interesting paths could be generated from a depth-one traversal of the complete Quicksort graph, except for two paths which are "hidden" because they cannot be derived locally. However, the set of interesting paths produced is sufficient to mode the program in the sense of assigning an unambiguous mode to all syntactic variables. This is not always the case!

Once we have generated a set of interesting paths, our algorithm proceeds by simply noting the modes of paths, first directly, and then by examining relationships between paths. There are essentially four different stages in the algorithm: 1) Assert absolute modes for some paths; 2) Assert that all paths on opposite sides of a body unification have opposite modes; 3) Proceed sequentially through the variables derivable from interesting paths, asserting all binary relations between paths, and 4) Repeatedly consider multiway relations (rule §3 Section 2) asserted by the clauses.

The first three stages have linear complexity. The multiway analysis is exponential in the number of variables, but by the time it is actually performed, most alternatives contradict the known modes, and thus are not explored. We found multiway analysis contributed only 2–7% of total analysis execution time in simple programs, and 11–20% in complex programs.

Important theoretical issues raised by this algorithm are its consistency, completeness, and safety. It is not difficult to prove that the finite domain algorithm is consistent in the sense that if, at some point in the analysis, path *p* is shown to have mode *m*, and if some subset of the interesting paths implies that *p* does not have mode *m*, then the algorithm will derive and report this contradiction. The major barrier to the consistency of this algorithm is somewhat subtle: the non-modedness of a program may not be detectable if the analysis uses the wrong set of paths! This leads directly to a reasonable definition of a *complete* set of paths. A set of paths generated for a program is *complete* iff the existence of a consistent moding for the set of paths implies that the program is well-moded.

Thus, the infinite set of all possible paths is a complete set; however, we are interested in *finite* complete sets and in particular in a *minimal* complete set of paths for the program. The path generation algorithm is incomplete; because of this incompleteness in path generation, the finite domain method is *unsafe*. It is a consequence of the incomplete set of generated paths that even if the program contains information about the mode of a path, that information may not be derived. Thus, the analysis is unsafe in the sense the compiler may not detect mode contradictions in erroneous (not well-moded) programs, and thereby produce erroneous mode information for programs that should be rejected altogether. Nonetheless, *most* generated paths in *typical* programs *are* moded by the finite domain analysis, and if the program being analyzed is known to be moded, all modes derived are correct. Assuming it can be made faster than safe analyses, unsafe analysis has utility for "lazy task creation" systems [8].

6 Performance Comparison and Conclusions

A benchmark suite of KL1 source programs (Table 1) was analyzed using the three mode analyzers. The analysis tools were all implemented in KL1 and run on the PDSS (V2.52.19) compiler-based system, on a Sun Sparcstation 10/30. PDSS executes about 34,000 reductions per second for the analyzers described here. The analyses tend to have complexity related to the number of symbols in the source program [14], which we categorize as constants (including functor symbols) and variable instances. Because paths can be cyclic, we define the number of "broken" paths, e.g., the **car** and **cdr** of a list will be counted, but not the **cadr** or **cddr**. We list, in parentheses, the number of paths produced by the finite-domain analyzer, since it may differ from the other three algorithms.

Execution performance is summarized in Table 2, giving the execution time (an average over five runs) and data memory consumption for each source program. The last row gives the static code size of the tools themselves. The broken path output of the analyses was verified as identical modulo the incomplete nature of the finite domain method. There are several interesting observations supported from the empirical measurements:

- Programs such as cubes and waltz contain ground lists of data that increase the analysis complexity by lengthening the propagated paths. Although we can hope that a ground data list of length one holds as much information as length 100,

program	proc	clause	symbols			broken paths	
			const	vars	total	total	avg length
qsort	2	5	31	40	71	19 (17)	1.6
primes	6	12	49	63	112	33 (28)	1.5
msort	4	11	54	75	129	36 (30)	1.7
queens	6	14	77	119	196	71 (43)	1.8
cubes	9	16	93	159	252	224 (79)	2.7
pascal	11	22	143	200	343	338 (56)	2.0
rucs	16	66	218	390	608	79 (46)	1.6
bestpath	20	44	279	492	771	507 (207)	2.5
waltz	20	54	333	630	963	329 ( )	2.2
waves	20	45	352	690	1042	623 (220)	3.0
triangle	42	80	315	1226	1541	1155 (648)	2.0

Table 1: Benchmark Suite Characteristics

benchmark	PDSS compile	finite domain	static graph	active graph	PDSS compile	finite domain	static graph	active graph
	exection time (msec)				memory consumption (kbytes)			
qsort	410	480	380	230	111	185	373	138
primes	780	950	610	370	164	320	616	204
msort	760	1,080	760	490	158	381	795	281
queens	1,140	2,190	1,210	620	244	692	1,444	340
cubes	1,570	3,030	1,730	1,080	320	1,057	2,070	571
pascal	1,660	4,710	2,410	1,080	343	1,362	3,272	574
rucs	3,010	7,970	6,370	2,350	699	1,838	8,224	1,343
bestpath	6,160	20,740	11,220	3,630	922	8,501	17,298	1,779
waltz	4,510	19,280	11,730	5,700	803	6,394	17,696	2,278
waves	7,960	37,540	16,320	4,690	1,204	9,098	28,371	2,318
triangle	11,720	52,920	51,440	9,660	1,865	17,339	51,656	4,437
analyzers					273	44	54	70

Table 2: Performance of Mode Analyzers (KL1 on Sun Sparcstation 10/30)

there is always an outside chance that the 99^{th} element will cause a contradiction somewhere in the program. We are considering methods wherein we can cut ground terms and then do post-analysis to ensure that we did not miss a contradiction.

- Development of the analyzers by novice programmers indicated some weaknesses of current CLP development environments. Notably, even after tuning, the finite domain analyzer is still generating excess suspensions and the static graph analyzer has high memory consumption. For example, although graph construction required 35% of total analysis time in the static graph analyzer, it was proportionally 68% in the faster active graph analyzer (written by an expert). This leads us to believe that the analyzers can be further tuned.

- Finite domain analysis does not appear faster than constraint propagation and therefore its utility is questionable. Although the reduction in paths decreases memory consumption slightly, it can in some cases produce *more* paths than are necessary, and in other cases delay resolution of multiway relations.

- The active graph analyzer demonstrates execution times ranging from 47% to 126% of the PDSS compiler. The arithmetic mean over the benchmarks is 69%, which we consider acceptable. For large programs, garbage collection remains problematic and thus we must throttle task creation. For example, analyzing triangle consumes 2.4 times the memory of the PDSS compiler.

We have described three alternative algorithms for rational-tree unification for the derivation of path modes in CLPs. We showed that mode analysis time was comparable to compilation time, which we consider reasonable, especially since Monaco [11], our native-code optimizing compiler (doing dataflow analysis, register allocation, etc.) has significantly slower compilation than PDSS. Currently the mode analyzers are being incorporated in an experimental FGHC-to-C compiler [7]; Morocco, a "lazy task creation" concurrent runtime system [8], and a thread partitioning compiler [1].

Acknowledgements

E. Tick was supported by an NSF Presidential Young Investigator award, with matching funds from Sequent Computer Systems Inc., and a grant from the Institute of New Generation Computer Technology (ICOT). Many thanks go to Bart Massey and Putthi Tulayathun for their research contributions.

References

[1] Z. M. Ariola, B. C. Massey, M. Sami, and E. Tick. Compilation of Concurrent Declarative Languages. Technical Report CIS-TR-94-05, University of Oregon, March 1994.

[2] M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inference. In *Int. Conf. and Symp. on Logic Programming*, pages 669–683. MIT Press, August 1988.

[3] T. Chikayama *et al.* Overview of the Parallel Inference Machine Operating System PIMOS. In *Int. Conf. on Fifth Generation Computer Systems*, pages 230–251, Tokyo, November 1988. ICOT.

[4] S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM TOPLAS*, 11(3):418–450, July 1989.

[5] S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. *Journal of Logic Programming*, 5(3):207–229, September 1988.

[6] A. King and P. Soper. Schedule Analysis of Concurrent Logic Programs. In *Joint International Conference and Symposium on Logic Programming*, pages 478–492. MIT Press, November 1992.

[7] B. C. Massey and E. Tick. Sequentialization of Parallel Logic Programs with Mode Analysis. In *LPAR'93*, number 698 in Lecture Notes in Art. Intell., pages 205–216, July 1993. Springer-Verlag.

[8] B. C. Massey and E. Tick. The Diadora Principle: Efficient Execution of Fine-Grain, Concurrent Languages. In *Hawaii Int. Conf. on System Sciences*, pages 396–404, January 1994. IEEE Press.

[9] E. Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[10] Z. Symogyi. A System of Precise Modes for Logic Programs. In *International Conference on Logic Programming*, pages 769–787. MIT Press, May 1987.

[11] E. Tick. Monaco: A High-Performance Flat Concurrent Logic Programming System. In *PARLE'93*, number 694 in Lecture Notes in Computer Science, pages 266–278. Springer Verlag, June 1993.

[12] E. Tick and M. Koshimura. Static Mode Analyses of Concurrent Logic Languages. Technical Report CIS-TR-94-06, University of Oregon, March 1994.

[13] E. Tick, B. C. Massey, F. Rakoczi, and P. Tulayathun. Concurrent Logic Programs *a la Mode*. In *Implementations of Logic Programming Systems*. Kluwer Academic Publishers, 1994.

[14] K. Ueda and M. Morita. Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, May 1994.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)
© 1994 IFIP. All rights reserved.

Demand-Driven Dataflow for Concurrent Committed-Choice Code

Bart Massey and Evan Tick

Dept. of Computer Science, University of Oregon, Eugene OR 97403, USA
{bart,tick}@cs.uoregon.edu

Abstract: Concurrent logic languages have been traditionally executed in a "greedy" fashion, such that computations are *goal-driven*. In contrast, non-strict functional programs have been traditionally executed in a "dataflow" fashion, such that computations are *demand-driven*. The latter method can be superior when allocation of resources such as memory is critical, which is usually the case for large, complex, and/or reactive programs. Specifically, demand-driven execution results in more efficient scheduling and improved termination properties. This paper describes a novel technique for demand-driven execution of concurrent logic language programs.

Keyword Codes: D.1.6, D.1.3
Keywords: Logic Programming; Concurrent Programming

1 Introduction

The difference between goal-driven and demand-driven execution of concurrent languages is a difference in execution focus: tasks or results? Goal-driven paradigms measure performance as *tasks executed per unit time* and eagerly schedule tasks with regard for little else. Demand-driven paradigms measure performance as *answers delivered per unit time* and lazily schedule tasks only when they are necessary to transform data needed to produce a result. The main advantages of goal-driven systems are simplicity of design and abundant parallelism. The main advantage of demand-driven systems is better resource allocation (e.g., memory usage) for resource-critical programs.

This latter advantage outweighs all others for programs that simply won't run otherwise. We illustrate this with an example comparing goal-driven execution of a concurrent logic (Strand [5]) program and demand-driven execution of a non-strict functional (Lazy ML [2]) program. Concurrent logic programming languages (which we will refer to as CCLs for *committed-choice languages* or *concurrent constraint languages*) are based on implicit parallelism which is exploited via synchronization on logic variables [9].

A program to find the first five odd integers by generate-and-test, written in Lazy ML and Strand, is given in Figures 1 and 2, respectively. The first program will almost immediately report the first five odd integers (in reverse order), whereas the latter one will always loop until it runs out of memory. The reason for this is that the traditional CCL execution model is goal-driven: when a clause's head tests succeed, all goals in the clause body are immediately candidates for execution. By contrast, the traditional

```
let rec generate n =
  n . generate (n + 1);

let rec test n (c . cs) l =
  if (n > 5) then
    l
  else (if (c % 2 = 1) then
    test (n + 1) cs (c . l)
  else
    test n cs l);

let five_odds =
  test 1 (generate 1) nil;
```

Figure 1: Sample Program in Lazy ML

```
generate( N, L ) :- true |
  N1 is N + 1, L := [ N | Ls ],
  generate( N1, Ls ).

test( N, _, L ) :- N > 5 | L = [].
test( N, [ C | Cs ], L ) :-
  N =< 5, C mod 2 =:= 1 |
  N1 is N + 1,  L := [ C | Ls ],
  test( N1, Cs, Ls ).
test( N, [C | Cs], L ) :-
  N =< 5, C mod 2 =\= 1 |
  test( N, Cs, L ).

five_odds( L ) :- true |
  generate( N, L1 ),
  test( 1, L1, L ).
```

Figure 2: Sample Program in Strand

model for non-strict functional languages is demand-driven: a function evaluation may be delayed until that function result is actually needed by the computation. Thus, the Strand program continues to **generate** more and more integers, whereas the functional program **generates** an integer only when the **test** demands one. When five odd numbers have been found, the program will immediately terminate.

The example is given in Lazy ML and Strand only to make it concrete. In general, *all* CCL implementations of which the authors are aware are goal driven. Most CCL implementations share the following characteristics: the compiler generates "goal stacking" code that creates goal records on the heap, instead of creating environment frames on a stack (as in sequential Prolog for instance). The goal records are managed as a pool or "ready queue (set)" from which "worker" processes can extract and add tasks. All bindings are made in a shared, global name space. A procedure invocation suspends when a required binding is not supplied by the caller. The required variable is linked to the suspending goal by some internal data structure, and later the goal is resumed if the variable is bound. This implementation scheme has evolved over the years and is resilient, but can be quite inefficient, as discussed in Section 2.

As Figure 1 illustrates, programs written in Lazy ML have nice synchronization properties. Unfortunately, the demand-driven implementation of Lazy ML and similar lazy functional languages is highly dependent on the purely functional nature of the language, and thus cannot easily support logical variables [7]. Therefore, the problem of demand-driven execution of CCL programs cannot be solved by a mere embedding of a CCL in Lazy ML — a new mechanism is needed.

Much work has been put into *mode analysis* of CCL programs. In its simplest definition, a mode of a variable occurrence in a procedure is "out" if the variable is bound in this procedure or its callees, and "in" otherwise [12]. Note that since variables can be bound to complex terms containing variables, in general we compute the modes of *paths* through terms to variables (at the leaves). There are several mode analyzers under development to collect this information (e.g., [10]), and we consider such analysis technology "a given" for this paper. We call a CCL program *fully-moded*, if (among other restrictions not relevant to this paper) there is at most one output occurrence of a variable in a clause body. We call the family of fully-moded concurrent logic programs FM for short. In an FM program, mode information always identifies the single occurrence of a variable at which the variable is bound (namely the body occurrence with output mode). It is this fact which makes demand-driven execution possible, thus, we limit our attention to FM in this paper.

This paper illustrates a new technique for demand-driven execution of FM programs, applicable to languages such as Strand [5], FGHC [9], and Janus [8]. The significance of the work is that it is the first specification of a purely demand-driven mechanism for CCLs (see discussion of related work by Ueda and Morita in Section 4). If successful, this mechanism can lead to a quantum performance improvement and will facilitate reactive programming.

2 Demand-Driven Evaluation

Because traditional CCL implementations do not rely on mode analysis, they must add all body calls of a clause to the ready set immediately upon execution of the clause. This leads to several sources of overhead relative to a demand-driven model, four of which are discussed here. 1) Body calls may be scheduled and executed even when the bindings they produce are not needed elsewhere in the problem. Even procedures which stop after producing a finite number of bindings often produce more bindings than are actually used. 2) Because producers of bindings may run arbitrarily far ahead of their consumers, resource exhaustion, particularly memory exhaustion, may cause a program to fail even though the language semantics imply its success. Thus, programmers have to be conscious of implementation details, and must often employ complex workarounds, such as bounded buffer techniques [5], to keep producers in check. 3) The ready set is accessed very frequently, and may get arbitrarily large. While workarounds exist for this problem, the ready set can nonetheless become a serious performance bottleneck. 4) Because a procedure may suspend, and indeed, may suspend upon several variables, a complicated system for suspension and resumption of procedures is necessary. This mechanism typically introduces high overheads for variable binding, as well as for the suspension and resumption itself.

We propose to solve these problems by implementing a demand-driven execution scheme for FM programs. At the heart of this scheme is the use of *continuations*, e.g., [1, 7]. A continuation consists of an environment, in this case a *frame pointer*, and a program point, or *instruction pointer*. Thus, an executing worker may save its current frame pointer and instruction pointer into a continuation. Later, that continuation may be *invoked* by loading an executing worker's frame pointer and instruction pointer from the continuation's, effectively continuing execution. Continuations have been used to great effect in programming language implementations [1], and have sometimes been made available to the programmer [3].

A reasonable way to implement demand-driven execution of FM programs, then, would be to do something analogous to dataflow-style execution: 1) When a value is needed for execution to proceed, the consumer of that value will ask the producer for the value, by invoking a continuation in the producer. 2) The producer will supply the value, by invoking a continuation in the consumer. Indeed, this is the first principle of our design: control flow should follow dataflow.

The problem then, is how to provide the consumer with a continuation by which it may obtain a value. The key is to realize that in logic programs, the value to be obtained is inevitably *the binding of a variable.* Indeed, in traditional CCL implementations, it is exactly this fact which is used to synchronize parallel execution: an invocation will suspend when a variable the consumer wants to read has not yet been bound. Thus, it is sufficient to make the following arrangement: at the time a logic variable is created, the variable's creator will bind the variable to a continuation which will produce that variable's value. 1) Since the producer and consumer of a variable's binding are known to share that variable, the consumer may bind to the variable the continuation which will consume the value produced, before invoking the continuation in the creator which will eventually produce the binding. 2) The creator will arrange for a value to be bound to the variable, usually by invoking a continuation in one of its body goals. 3) The producer will bind the variable to a value before invoking the continuation which will consume the value. This is the second principle of our design: consumers and producers of a variable's binding should communicate through that variable.

The implementation should allow parallel execution, but the execution model outlined so far is sequential. To understand where the parallelism comes from, and the machinery needed to handle it, it is necessary to understand some details that have been so far avoided. First, where does the binding of a variable to a producer continuation come from? The answer is that when the variable is *created*, it is allocated in the frame of the invocation creating it, and is initialized to a continuation in this invocation. When this continuation is invoked, as noted above, there exists enough information to determine which body call will be necessary to produce a value binding for the variable.

Second, suppose that multiple consumers try to read the same variable before a binding for it is produced? This can easily happen on parallel hardware: while one worker is in the midst of producing a binding for a variable, other workers try to read the variable. The answer to the question is twofold: the variable must always be labeled with a tag indicating whether a binding is currently being produced for it, and if a binding is pending the consumer continuation must be added to a *set* of continuations bound to the variable. The producer can invoke one of the continuations directly upon finally producing a value binding, but other workers searching for work must have access to the remaining continuations, which implies a global ready set. Thus while a global ready set is apparently inevitable, a global suspend set can be avoided, by enqueuing suspended threads of execution directly on the variable causing suspension, as continuations. The third principle of our design might be stated by analogy with an epigram attributed to Einstein: avoid global objects as much as possible, but not more so.

3 Implementation Model

Having sketched the principles of operation for demand-driven execution of FM programs, we now discuss the technical details. Several preconditions must be met to make an FM program suitable for the execution model: 1) each procedure will have the head and guards of its clauses "flattened" and formed into a decision graph [6] which will select a clause body for execution; 2) each clause body will be partitioned into a "tell" part, which contains all body unifications, followed by a "call" part, containing all body calls; 3) similarly, each output argument of a clause will be categorized as either a "tell" output argument, meaning that its binding is produced locally by a tell binding, or a "call" argument, meaning that its binding is produced by a body call; 4) the binding occurrence of each variable in the clause body will be identified.

The execution model also has ramifications for FM semantics. Since execution is now demand-driven, the notion that a computation terminates when there are no ready goals no longer applies. For simplicity's sake, the computation will terminate when all output arguments of the query have been bound to ground terms.[1]

[1] If more complicated query termination conditions were required, new query syntax could be introduced to achieve them.

Global State: The only global object required by the model is the ready set **ready**. This will be implemented as a locked queue of *active messages* [4, 11]. A *message* (for short) is a tuple <**continuation**, **contents**>, where the **contents** of the message is merely some value to be communicated: the **mr** register is loaded from the **contents** during message execution.

Worker State: The worker state consists of several abstract registers. The "instruction pointer" **ip** points to the next instruction to be executed. The "frame pointer" **fp** points to the context in which to execute. The "message register" **mr** is used for communication across continuations. The "argument pointers" $\mathbf{ap}_0..\mathbf{ap}_n$ are used to pass arguments to an invocation. The "call register" **cr** is used during frame creation.

Variables: A variable is a locked tuple <**state**, **binding**>, where the **state** is an element of the set {**unread**, **read**, **written**}, and the **binding** is either a continuation, a continuation list, or a value, depending on the **state**. The state always begins as **unread** and increases monotonically to **written**: this corresponds to the single-assignment property.

Frames: A *frame* is the environment of a particular procedure invocation, and as such, must contain all invocation-specific information. A frame is a tuple containing the following:

- **commit_index:** The index of a clause which the invocation has committed to, or **nil** if the procedure has not yet committed.
- **suspcount:** The number of variables which must be bound before decision graph evaluation can continue.
- **tobind:** A list of *binding indices*, which will be used to restart other suspended calls to this invocation after commitment.
- **params:** A save area for passed parameters of the invocation.
- **vars:** A tuple of variables "created" by the invocation.
- $\mathbf{callfp}_i$**:** The $\mathbf{callfp}_i$ fields are used only after commitment: $\mathbf{callfp}_i$ is either a locked pointer to the frame created for the i^{th} body call of the invocation, or **nil** if no frame has yet been created for this call.
- **locals:** A scratch area, used for objects which will not be visible outside the invocation.

Figure 3: Objects Required by the Model

The demand-driven nature of execution allows a new guarantee about program execution: any deterministic program which *could* complete in a finite number of steps in a traditional implementation *will* complete in a finite number of steps in this implementation. Further, *any* program which will complete in a finite number of steps in this implementation will complete having created a *minimal* number of procedure invocations. These guarantees are basically a promise to the programmer that scheduling considerations are not part of programming for the demand-driven implementation.

We characterize the objects required by the model in terms of their scope: four kinds of state which are relevant (Figure 3). The *global state* is visible and accessible everywhere during execution. The *worker state* is information available only to a particular worker during its execution. *Variables* are available both to their creator and to any procedure invocation which has received them as parameters. *Frames* are available only to a particular procedure invocation. The execution mechanism is given in Figure 4 and illustrated in Figure 5. The heavy lines indicate parent-child invocation relationships, with the dashed lines indicating possible intervening invocations. The light lines indicate message passing. Entry points **read, resume, bind,** and **create** are defined later in this section.

1. An invocation, which we will call the "consumer" of a value, needs a variable to be bound in order to proceed with execution. This must be the result of the fact that the consumer needs the binding in order to commit to a clause.
2. The consumer obtains from the variable a continuation in another invocation, which we will call the "creator" of the variable being bound. This continuation will have the creator's **fp**, and the **read** entry point of the creator's procedure as the **ip**.
3. The consumer changes the **state** of the variable from **unread** to **read**, and makes the **binding** of the variable a **resume** continuation in the consumer, which will utilize the bound variable when it becomes available.
4. The consumer places a pointer to the variable in the **mr** of the worker, and invokes the **read** continuation.
5. The creator of the variable determines which body call is needed to bind the variable. It then computes a "binding index" **bi** denoting which output argument of the call is being requested, and places the binding index in **mr**.
6. If the creator has previously created the frame necessary for the body call, it loads **fp** with a pointer to this frame, and enters at the **bind** entry point of the procedure being called. Otherwise, the creator allocates a new frame, loads **fp** with a pointer to it, and enters at the **create** entry point of the procedure being called.
7. The previous step is repeated until an invocation, which we will call the "producer," actually produces a binding for the variable.
8. The producer obtains from the variable the **resume** continuation in the consumer which will consume the value.
9. The producer rebinds the variable so that its state is **written** and its **binding** is the variable's value.
10. The producer invokes the saved **resume** continuation, and the consumer uses the bound value.

Figure 4: Demand-Driven Execution Mechanism

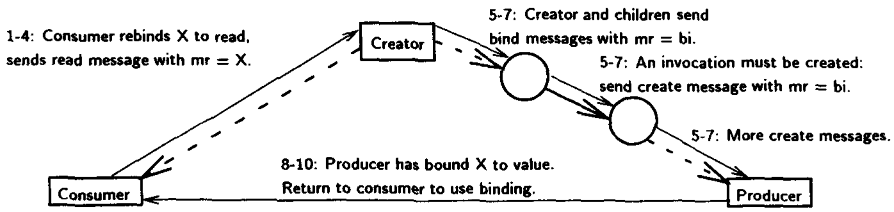

1-4: Consumer rebinds X to read,
sends read message with mr = X.
Creator
5-7: Creator and children send
bind messages with mr = bi.
5-7: An invocation must be created:
send create message with mr = bi.
5-7: More create messages.
8-10: Producer has bound X to value.
Return to consumer to use binding.
Consumer
Producer

Figure 5: Control and Data Flow During Model Execution

The model, however, is complicated by the concurrent semantics of the language, and by the desire to exploit parallelism in the implementation. There are several places during execution where more than one thread of execution may need to suspend until a condition is satisfied, and all of these threads may be resumed in parallel. In addition, there is one place in which new threads of execution need to be created, and may be started in parallel.

The first place in which threads may need to suspend is the consequence of the fact that, between the time that a variables state goes to **read** and the time it goes to `written`, other procedures may also need to read the variable's value. The protocol described above ensures that only one thread will try to produce the value, but it is also necessary to ensure that, once the variable's value is `written`, all threads which were waiting for the value will resume. To this end, the variable's binding when in the **read** state is a *list* of continuations which should be placed on the ready queue when the variable's value becomes available. (Note that the ready queue is a message queue, not a continuation queue. Note also that the message register `mr` is not used during resumption of suspended readers.)

The fact that, before a procedure invocation has committed to a particular clause body, several outputs of that procedure may be requested implies that threads must be suspended until commitment is complete, then resumed. This is the purpose of the `tobind` element of the frame: it is used to hold a list of binding indices which have been requested by callers. When commitment is complete, each binding index is packaged up as the `value` portion of a message whose `ip` points to code in the procedure which will bind the variable, and whose `fp` points to the invocation's frame. These messages are then placed on the ready queue, so that all of the bindings may proceed in parallel.

New threads are created as the result of *strict* operations during commitment, i.e., operations which must have all argument values in order to proceed. An example of a strict operation is the arithmetic operation +/2. This operation must have *both* argument values before computing the result. Since all arguments are required, they should be produced in parallel. Thus, messages which will produce all bindings are placed on the ready queue. The frame's `suspcount` is used to keep track of when all required bindings have been obtained, at which point execution may resume.

The execution model, now complete, is fully specified in Figure 6. Due to limited space, we omit details concerning locking and variable update. Each procedure has three entry points: **read**, **create**, and **bind**.

- The **read** entry point is always reached through a continuation, when a consumer attempts to bind an `unread` variable. The message register `mr` will contain a pointer to the variable to be bound. The **read** procedure ensures that a variable's binding will be produced, and arranges for execution to resume in the consumer thereafter.

- The **bind** entry point will be called in order to bind an output parameter of the procedure. The message register `mr` will contain a binding index indicating the parameter to be bound. **read** performs body calls necessary to obtain bindings for requested output arguments.

- The **create** entry point will be called in order to create a procedure invocation (typically by the creator of a variable). The frame pointer `fp` will point to an uninitialized frame. The message register `mr` will contain a binding index. The call register `cr` will contain a pointer to the caller's $\texttt{callfp}_i$. The **create** procedure handles invocation initialization, and then starts the commitment process, arranging to **bind** requested output arguments as soon as commitment completes. Decision graph evaluation may have to **suspend** waiting for bindings. If it does, it will **resume** when bindings are available.

```
To switch threads:
  • Dequeue a message m from the ready queue. Load m.mr and m.fp. Jump to m.ip.

To read a variable V passed in mr in a procedure p:
  • If mr.state is written:
      • Switch threads.
  • If mr.state is read:
      • Find the binding index bi denoted by mr. Place bi in mr. Invoke p.bind.

To bind an output argument of a procedure p:
  • If mr refers to a tell argument:
      • Switch threads.
  • If mr refers to a call argument:
      • Find the procedure q and binding index bi denoted by mr. Find the call index ci of
        q. Place bi in mr.
      • If fp.callfp$_{ci}$ is non-nil:
          • Invoke q.bind.
      • If fp.callfp$_{ci}$ is nil:
          • Set up the arguments of a call of q. Allocate and set up a new frame in fp. Invoke
            q.create.

To create a new frame and bindings in a procedure p:
  • Initialize the frame's commit_index, tobind, var, and callfp$_i$. Place fp in cr.fp, marking
    the frame as created. Jump to the decision graph for p.

To suspend decision graph evaluation:
  • Set fp.suspcount to the number of variables being suspended on.
  • For each variable V whose binding is needed:
      • If V.state is unread:
          • Save the binding of V and a pointer to V as a message m. Mark V read, and add a
            resume continuation to V's binding. Add m to the ready queue.
      • If V.state is read:
          • Add a resumption continuation to V's binding.
  • Switch threads.

To resume a suspended decision graph:
  • Decrement fp.suspcount. If fp.suspcount is non-zero, switch threads. If fp.suspcount
    is zero, resume decision graph execution.

To commit to a particular clause:
  • Set fp.commit_index to the desired clause.
  • For each tell output argument V:
      • If V.state is read:
          • Save the binding of V in some temporary register r. Bind V to its value.
          • For each continuation c in r:
              • Add a message containing c to the ready queue.
      • If V.state is unread:
          • Bind V to its value.
  ◇ For each call output argument V:
      ◇ If V.state is unread:
          ◇ Set V.binding to p.read.
  • For each non-tell binding index bi in fp.tobind:
      • Add a message <fp, p.bind, bi> to the ready queue.
  • Switch threads.
```

Figure 6: Procedures for Demand-Driven Execution

4 Discussion

There are a number of optimizations possible in the basic execution model. First, there are several places during execution in which messages are placed on the ready queue, and then a thread switch is done. Instead, some ready queue traffic can be avoided if, in the spirit of the original idea, one of the messages is selected for direct invocation rather than enqueuing and then immediately dequeuing it. Second, rather than adding messages to the ready queue individually, some efficiency can be gained by adding the entire batch at once. Indeed, if appropriate data structures are used, this may be almost as cheap as enqueuing an individual message. Third, although locking is not specified at this level of detail, it is apparent that much locking can be avoided due to the monotonic progression of such operations as variable binding and commitment — a thread can check to see whether a lock is necessary and avoid the lock if not. Fourth, since the **mr** is unused during resumption, it may be used by the producer to return the requested variable's binding to the consumer, avoiding an unnecessary variable reference.

The portion of the **commit** description labeled "◇" in Figure 6 is optional: it is unknown whether execution will be more efficient with it, since some indirections are potentially avoided, or without it, since it implies extra locking and variable binding.

There are several possible criticisms of the execution model. We discuss three. 1) Nondeterministic programs may invoke procedures which do no useful work. But *any* implementation must schedule these procedure calls: only an oracle could tell which invocation will produce the bindings necessary for nondeterministic execution to proceed. 2) Because of the change in termination conditions to achieve demand-driven execution, a few existing CCL programs might not run correctly under this model. In particular, some programs may deadlock or livelock because they expect to produce non-ground query outputs. However, the changes needed to make existing programs operable should be straightforward. Any slight incompatibility is outweighed by the fact that scheduling and throttling of producers is no longer a concern, which should make it considerably easier to write new code for this implementation. 3) Compilation details such as the storage of temporaries in registers are outside the scope of this paper. However, because of the frequent switching of environments during execution, it may be difficult to fully utilize the large register set of modern CPUs for efficient temporary storage. The efficiency increases due to demand-driven execution should outweigh this loss.

Historically, CCLs sacrificed backtracking in order to achieve efficiency. A consequence was the elimination of all speculative *or-parallelism*: a worker will not attempt to execute a given clause unless that clause will contribute bindings needed to answer the query. Analogously, our demand-driven implementation eliminates all speculative *and-parallelism*: a worker will not attempt to execute a given body call unless that call will contribute bindings needed to answer the query. This throttling of all speculative parallelism can lead to great efficiency in problem solution, but it may not lead to the fastest solution.

The only work related to our proposed scheme of which we are aware is that of Ueda and Morita [11], who describe a model which uses active messages to improve the performance of producer-consumer stream parallelism. Our use of active messages, as well as some of our fundamental philosophy, is essentially the same as theirs. However, their method has important differences from ours. First and foremost, Ueda and Morita's technique still is producer-driven rather than consumer-driven: it attempts only to optimize the overhead of producer-to-consumer communication. Thus, none of the benefits of automatic demand-driven execution accrue (although some methods of using programmer annotations to inhibit producers outrunning consumers are discussed). Second, their technique requires a sophisticated type analysis in addition to mode analysis. Finally, it is an optimization for certain limited situations only, in otherwise conventional execution.

5 Conclusions

We have presented a novel execution model for flat concurrent logic programming languages (CCLs), based on mode analysis and on continuation passing utilizing shared variables. This model achieves demand-driven execution of CCLs, which at once achieves greater execution efficiency and simplifies programming. This paper serves as a schematic for those wishing to build high-performance demand-driven CCL systems. We hope to begin work soon on an exploratory implementation of the technique, in order to gain insights into its benefits and drawbacks.

Acknowledgements

This research was supported by a grant from the Institute of New Generation Computer Technology (ICOT) and an NSF Presidential Young Investigator award, with matching funds from Sequent Computer Systems Inc.

References

[1] A. W. Appel, editor. *Compiling With Continuations.* Cambridge University Press, 1992.

[2] L. Augustsson and T. Johnsson. *Lazy ML User's Manual.* Available via FTP from **animal.cs.-chalmers.se**, June 1993.

[3] W. Clinger and J. Rees (ed.). Revised^4 Report on the Algorithmic Language Scheme. Technical Report CIS-TR-91-25, University of Oregon, Department of Computer Science, February 1992.

[4] T. von Eicken *et al.* Active Messages: a Mechanism for Integrated Communication and Computation. In *Int. Symp. on Computer Arch.*, pages 256–266. IEEE Computer Society Press, May 1992.

[5] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice Hall, 1989.

[6] S. Kliger. *Compiling Concurrent Logic Programming Languages.* PhD thesis, The Weizmann Institute of Science, Rehovot, October 1992.

[7] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall International Ltd., 1987.

[8] V. A. Saraswat, K. Kahn, and J. Levy. Janus: A Step Towards Distributed Constraint Programming. In *North American Conference on Logic Programming*, pages 431–446. MIT Press, October 1990.

[9] E. Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[10] E. Tick. Practical Static Mode Analyses of Concurrent Logic Languages. In *International Conference on Parallel Architectures and Compilation Techniques*, Montreal, August 1994. North-Holland.

[11] K. Ueda and M. Morita. Message-Oriented Parallel Implementation of Moded Flat GHC. In *Int. Conference on Fifth Generation Computer Systems*, pages 799–808, Tokyo, June 1992. ICOT.

[12] K. Ueda and M. Morita. Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, May 1994.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Exploitation of Fine-grain Parallelism in Logic Languages on Massively Parallel Architectures

Hiecheol Kim and Jean-Luc Gaudiot

Department of EE-Systems, University of Southern California, Los Angeles, CA 90089-2563, USA

Abstract: Logic programming has been drawing increasing attention as a parallel programming paradigm due to its ease of programming. However, an important question remains: Can logic programs achieve scalable performance on general-purpose massively parallel architectures? The problem is to determine how to efficiently incorporate parallelism in logic programs within the architectural features of such machines. We frame the question by identifying important issues that are crucial for scalable performance. We have systematically developed the logical organization of a parallel execution model based upon data-driven principles of execution. In this paper, we present some details of the model, particularly its fine-grain parallelism support and the self-scheduled inference as the basis for its distributed implementation.

Keyword Codes: D.1.6
Keywords: Logic Programming

1 Introduction

Logic programming based on universally quantified Horn clauses is becoming an accepted programming paradigm for symbolic computing. PROLOG is indeed one of the most popular logic programming languages because of its many advantages in terms of ease of programming and declarative semantics. The advent of massively parallel architectures, along with the inherent opportunities for parallelism in logic languages, should allow scalable performance during the execution of logic programs.

However, previous effort on the parallel execution of logic programs focused mostly on shared memory multiprocessors [1]. With these machines, it is hard to expect performance increasing proportionally to the number of processors since logic programs make heavy use of the shared memory which then becomes a bottleneck [2]. Instead, considerable effort has recently been made for the implementation with distributed memory multiprocessors [4, 5, 7]. In order to fully materialize the expected scale-up in performance on the architectures, more study based on real implementations is, however, still needed.

In the research reported in this paper, we have selected message-passing, distributed memory multiprocessors in the hope to reach a scalable performance of our logic programs. Further, each processor may possess additional hardware support for dynamic synchronization. The memory is organized in a non-single address space. The latency of local *vs.* remote memory accesses typically differs by several orders of magnitude, (the remote memory latency grows with the size of the machine while the local access time remains more or less constant).

2 Parallel Execution of Logic Programs

The semantics of logic languages, *i.e.*, the principle of SLD refutations, entail the exploration of a search tree at runtime for scheduling and environment stacking reasons. The globally shared property of the tree could render inefficient the parallel implementation of logic languages on machines with a non-single address space, which is obviously an acute problem in the distributed memory multiprocessors we are targeting for this work. In this section, we summarize the three important issues to be addressed if we are to achieve scalable performance of logic programs in distributed memory multiprocessors. We further discuss our approach to solving the problems.

2.1 The Issue of Scalable Performance on distributed memory multiprocessors

Distributed Scheduling: In shared memory systems, runtime scheduling can be performed by a single centralized scheduler with a single copy of the search tree shared by all PEs. The many advantages of this scheme cannot be transferred to distributed memory multiprocessors due to their large system size and their memory structure. Scheduling thus becomes one of the important issues when implementing logic programs. Some form of distributed scheduling must therefore be implemented to observe performance that is scalable. As for the search tree, it will be more advantageous to distribute it across the nodes rather than placing it on a single node. Precise partitioning strategies will be discussed later.

Multiple Active Tasks on a PE: The traversal of a shared search tree in search of environment variables contained in other nodes may cause low system utilization due to the remote accesses. To avoid these remote accesses, most other approaches have recourse to binding methods which force memory accesses to be local in a PE. However, the methods may impose an intolerable overhead with additional computations such as environment closing and structure copying. It is the thesis behind our approach that the idea of overlapping the computation with the communication caused by remote accesses is valuable for logic programming on distributed memory multiprocessors as well as for conventional imperative programming. From an architectural viewpoint, the approach to parallel execution based on a single "active" thread in a PE [3] is not suitable for some promising multiprocessors such as data-flow or multithreaded machines due to insufficient fine-grain parallelism within a single thread. As opposed to a sequential single threaded execution, we will therefore aim at maintaining multiple "active" tasks simultaneously in a PE for the interleaved execution of fine-grain parallelism.

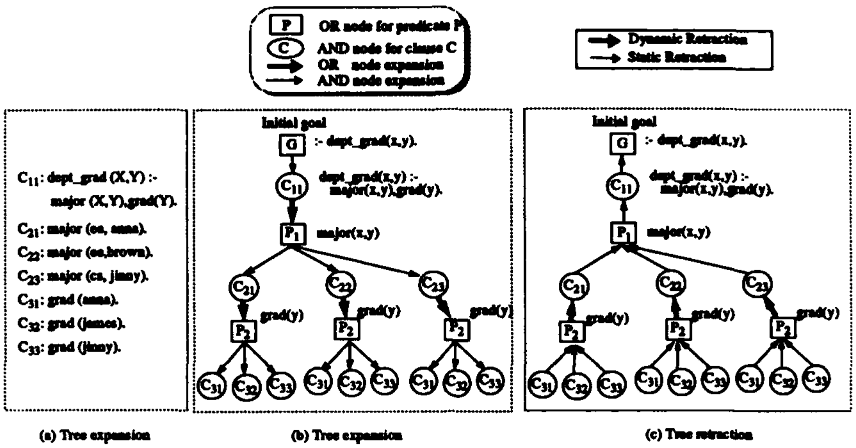
Exploitation of Fine-Grain Parallelism: The exploitation of fine-grain parallelism for multiprocessors is obviously crucial as a means to hide memory latency. In addition to *unification parallelism*, there exist more opportunities for fine-grain parallelism in logic languages. They are *generation parallelism* obtained by preparing all the arguments of a goal in parallel in the argument frame and *instantiation parallelism* uncovered by instantiating all the components of a complex term in parallel when instantiating a complex term (*e.g.*, a list or a structure) in the heap during argument generation or unification. It is very expensive to exploit unification parallelism due to the requirement for garbage collection, because the failure to unify any one pair of arguments makes the unification of other arguments being done in parallel useless. Thus, our model exploits unification parallelism only for the head literals that always yield successful unification, whereas it exploits generation and instantiation parallelism with only a modest alteration of conventional logic engines.

2.2 The Non-deterministic Data-Flow (Ndf) Model

As one solution to the above problems, we propose a parallel execution model which includes architectural support for dynamic synchronization, such as data-flow and multithreaded architectures [6]. In our model, the runtime behavior of logic programs, *i.e.*, search tree expansion and retraction, is modeled into the data-driven execution. The data-flow graph representation for logic languages thus obtained is called the Non-deterministic Data-Flow (NDF) graph. The principle of data-driven computation enables the inference of logic programs to be scheduled at runtime in a distributed fashion.

Semantics of the (NDF) Graphs: The NDF graph retains the "conventional" data-flow graph arcs, node firing upon data availability, etc. The graph can be considered a tagged token dynamic data-flow graph in that tokens are tagged to carry information about dynamic instances of the actors. As the two unique characteristics of the NDF graphs, multiple incoming arcs are allowed to any one input port of a node and an undetermined number of tokens with the same color are allowed on an arc at the same time (hence its name, Non-deterministic Data-Flow (NDF) graph). Therefore, an arc in an NDF graph is either a *deterministic* or a *non-deterministic* arc. Only a single token of one given color is allowed on a deterministic arc, whereas multiple tokens of the same color may exist simultaneously on a non-deterministic arc.

The NDF Graphs Representation: For logic languages which consist of Horn clauses, *i.e.*, definite clauses and goals, the NDF graph is composed of the following principal classes of reentrant graphs; (1) A predicate graph is defined for each predicate since they are invoked globally in a program. (2) A clause graph is defined for each clause for flexibility in scheduling and allocation, although the scope of a clause is limited inside the predicate to which the clause belongs. (3) Module graphs are defined for a set of common modules that occur in predicates or clause graphs. Each graph comprises nodes with one input port, two output ports and its graph body. The arcs connected to the ports are assigned with special roles with respect to token types and properties with respect to token colors; *Input arcs* are deterministic and activation (AT) tokens on the arcs serve to activate either a predicate or a clause graph. The tokens are comprised of: 1) a mode field to specify the execution mode of an activation (sequential or parallel mode), 2) pointers to an argument frame and an environment, and 3) a backward-continuation field prepared for the execution control such as backtracking. *Solution arcs* are paths for solution tokens yielded from the evaluation of a graph. As multiple solutions may be generated, these arcs are non-deterministic. A solution (SL) token contains an environment which incorporates the new bindings done at evaluation time. *Control arcs* are deterministic arcs and tokens on them have a special role in execution control. For example, failure (FL) tokens serve to report the failure of a clause. They are utilized in detecting the backtracking condition of a predicate node to which the clause belongs. Backtrack (BT) tokens serve to report the failure of a predicate. Completion (CM) tokens serve to report that a predicate has completed its evaluation.



(a) Tree expansion

(b) Tree expansion

(c) Tree retraction

Figure 1: An example of the tree expansion and retraction

3 Data-Driven Self-Scheduled Execution

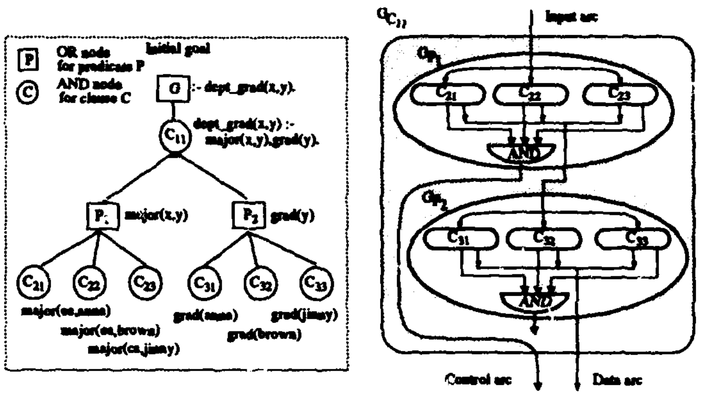
One special behavior of logic programs is that a search tree is expanded or retracted as the computation progresses. Before we proceed with the data-driven execution mechanism, we thus present a set of definitions related to the expansion and retraction of the runtime search tree.

As for the tree expansion, we define two types; OR-*node expansion* denotes the expansion due to the activation of an OR-node. This type of expansion is depicted by a thick arrow in Fig. 1 (b). AND-*node expansion* denotes the expansion due to the activation of an AND-node. This type of expansion is depicted by a thin arrow in Fig. 1 (b).

A search tree is retracted along the direction opposite to that of tree expansion when an active OR- or an AND-node has failed. Given two nodes involved in a retraction, the destination node of the two is not always determined at compile time. We thus say that two nodes involved in a retraction are in a ***static retraction*** relation if the destination can be statically determined; otherwise, they are in a ***dynamic retraction*** relation. In an AND-OR tree, an AND-node and its parent OR-node are in a static retraction relation. Nodes C_{21} and P_1 in Fig. 1 (c) are an example of this case. An OR-node for the first goal in a clause and the AND-node for the clause are also in a static retraction relation. P_1 and C_{11} in Fig. 1 (c) are an example of this case. On the other hand, in an AND-OR tree, any non-leftmost child OR-node of a parent AND-node and the terminal AND-nodes expanded from a predecessor child OR-node are in a dynamic retraction relation. The retraction from P_2 to C_{21} in Fig. 1 (c) is one case of the dynamic retraction.

3.1 Clause NDF graphs and the AND-OR trees

Suppose that a clause contains n goals in its body. The NDF graph for the clause will be G_c and the predicate graph for the i^{th} goal will be G_{p_i}. The NDF graph G_c contains n graph linkages for predicate graphs G_{p_i} $(1 \leq i \leq n)$. The input and solution arcs are



(a) The AND/OR Tree

(b) The NDF graph for C_{11}

Figure 2: Example: an NDF graph for a clause

connected such that the input arc of G_c becomes that of G_{p_1}, the solution arc of G_{p_i} becomes the input arc of $G_{p_{i+1}}$, and the solution arc of G_{p_n} becomes that of G_c.

The control arc of G_{p_1} is connected to the control output port of G_c. That is, the BACKTRACK token from the predicate graph for the first goal results in the production of a FAILURE token in the clause graph. The control output port of G_{p_i} $(1 < i \leq n)$ remains unconnected. However, the predicate graph changes the tag value of the control token on the port so that the control token will be dynamically directed to its destination. The details will be covered in later sections.

Fig. 2 (b) depicts the inside of an NDF graph for clause C_{11} in the example program and the relation between the clause graph and its corresponding AND-OR tree. The NDF graph for clause C_{11} is equivalent to a subtree whose root node is the AND-node for C_{11} in the AND-OR tree depicted in Fig. 2 (a). Again, the NDF graph for a predicate is a subtree whose root node is the OR-node for the predicate in the AND-OR tree.

3.2 Forward Propagation: Tree Expansion

Forward propagation corresponds to the propagation of tokens for the purpose of realizing a tree expansion in the NDF model. It is supported by input and output solution arcs in the NDF graphs and consists of a series of relevant (predicate or clause) graph activations by AT tokens on the arcs.

OR-node expansion: Inside a clause graph G_c, a predicate graph for the first goal is activated by an input AT token. Each SL token yielded from the activation of a predicate graph leads to the activation of the predicate graph for the next goal. Here, the activation corresponds to OR-node expansion in the AND-OR tree. In Fig. 3, the AT tokens issued by $G_{C_{21}}$, $G_{C_{22}}$, and $G_{C_{23}}$ inside G_{P_1} trigger the three instances of the predicate graph G_{P_2} that correspond to an OR-node expansion.

AND-node expansion: The input AT token of a predicate graph G_p comes to trigger the activations of the clause graphs inside G_p, each of which corresponds to the AND-node expansion in the AND-OR tree. In Fig. 3, the three activations of $G_{C_{31}}$, $G_{C_{32}}$, and $G_{C_{33}}$ inside the predicate graph G_{P_3} triggered by any token yielded from the activation of the previous goal G_{P_2} correspond to an AND-node expansion.

3.3 Backward Propagation: Tree Retraction

Backward propagation denotes the propagation of tokens between two graphs that correspond to two nodes either in a static or in a dynamic retraction relation in the AND-OR tree. The control token, thus propagated, causes a specific action according to the type of the token, *e.g.,* backtracking or program termination detection. The following outlines how the backward propagation is implemented and how the tree retraction is realized with it.

Static Retraction: Backward propagation is done via a statically provided control arc between a source and its destination NDF graph. Indeed, Fig. 2 (b) shows that control arcs exist (i) from G_{p_1}, the predicate graph for the the first goal of C_{11}, to its parent clause graph $G_{c_{11}}$ and (ii) from clause graphs ($G_{c_{21}}$, $G_{c_{22}}$ and $G_{c_{23}}$) to their parent predicate graph G_{p_1}.

Dynamic Retraction: In this case, the destination graphs cannot be determined at compile time. Backward propagation is implemented by retagging control tokens as follows; (1) When the activation of a unit clause graph yields a solution, the continuation[1] of the subgraph, that serves backward propagation, in the graph is placed in the backward-continuation field of the SL token. (2) The activation of the first predicate graph caused subsequently by the SL token extracts the continuation from the token and retains it. (3) When a control token from a child graph reaches the subgraph that serves backward propagation in an active predicate graph, the subgraph will take an appropriate action according to the type, *e.g.*, backtracking. It will then retrieve the continuation extracted at step 1, form a new control token by tagging it with the continuation, and place the token on the output control port. The control token will be routed to the subgraph inside the unit clause in step 1 that created the SL token.

In order to clarify the above mechanism, we now show how the backward propagation operates to perform backtracking on the example program.

- **Step (i)**: In Fig. 3, each SL token, issued respectively by $G_{C_{21}}$, $G_{C_{22}}$ and $G_{C_{23}}$ carries a continuation in which a continuation is expressed simply by a clause name in the token.

- **Step (ii)**: The SL tokens activate three instances of G_{P_2} in Fig. 3. The continuations are extracted and kept in each activation, which is graphically indicated in a small circle beside G_{p_2} in Fig 3.

- **Step (iii)**: Inside G_{P_3}, three clause graphs, $G_{C_{31}}$, $G_{C_{32}}$ and $G_{C_{33}}$, are activated by the token labeled C_{21}. Suppose that every activation fails. Then, each activation issues a FAILURE token, *e.g.,* f_{31}, f_{32}, f_{31}. The propagation of FAILURE tokens corresponds to a static retraction, as indicated in Fig. 3.

- **Step (iv)**: The predicate graph then retrieves the continuation extracted in step (ii) and produces a BACKTRACK token bt_{21} in which 21 indicates the destination

[1] Continuation of a graph refers to a tag which consists of the color used for the activation and the address of the graph.

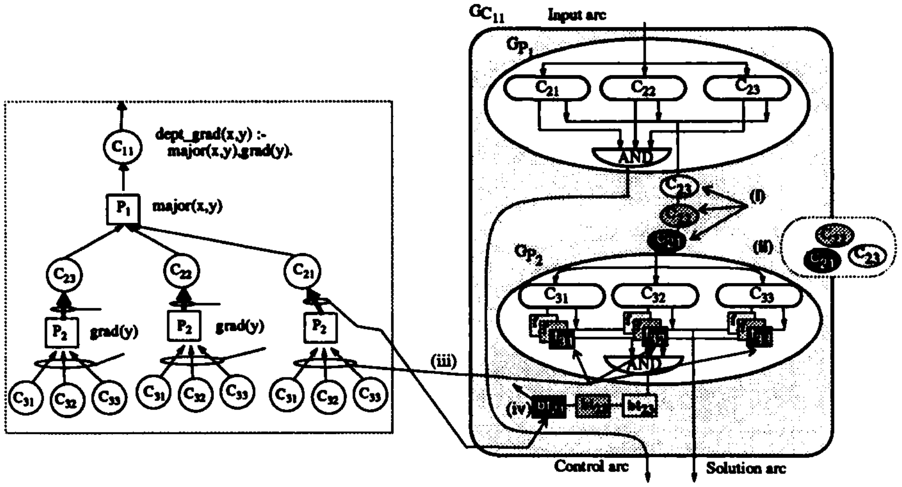


Figure 3: Example of backward propagation (1)

graph in the continuation, *i.e.*, C_{21}. Bt_{21} is then propagated to a subgraph inside $G_{C_{21}}$, which in turn corresponds to a dynamic retraction as indicated in Fig. 3.

- **Step (v)**: The BACKTRACK token sent to $G_{C_{21}}$, in turn, produces a FAILURE token f_{21} in Fig. 4. Similarly, f_{22} and f_{23} are supposed to be generated through Step (iii) and Step (iv) for $G_{C_{22}}$ and $G_{C_{23}}$.

- **Step (vi)**: Fig. 4 shows that the predicate graph G_{P_1} generates a BACKTRACK token bt_1 upon f_{21}, f_{22} and f_{23}. The propagation of bt_1 in Fig. 4 corresponds to a static retraction since P_1 is the first goal in C_{11}.

- **Step (vii)**: The BACKTRACK token bt_1 will be propagated out of the $G_{C_{11}}$ as a FAILURE token f_{11} in Fig. 4. The propagation corresponding to a static retraction indicates that the execution of the program has finally failed.

4 Fine-Grain Parallelism Support in the NDF model

Parallel execution models of PROLOG mostly employ inference engines based on the Warren Abstract Machine (WAM), the storage model of which is designed under the assumption of a single active task within each PE. With this storage model, the interleaved execution of multiple active tasks is inherently very difficult due to its stack-based organization. Also, some instructions (*e.g.*, unify-*xxx*) add the complexity of the interleaved execution because they operate differently depending on the mode (*read* or *write*) set to the CPU [8]. Therefore, in order to exploit fine-grain parallelism, it is required to alter the conventional storage model and to modify the above mode-sensitive instructions. In this section, we briefly explain the support for fine-grain parallelism in the NDF model.

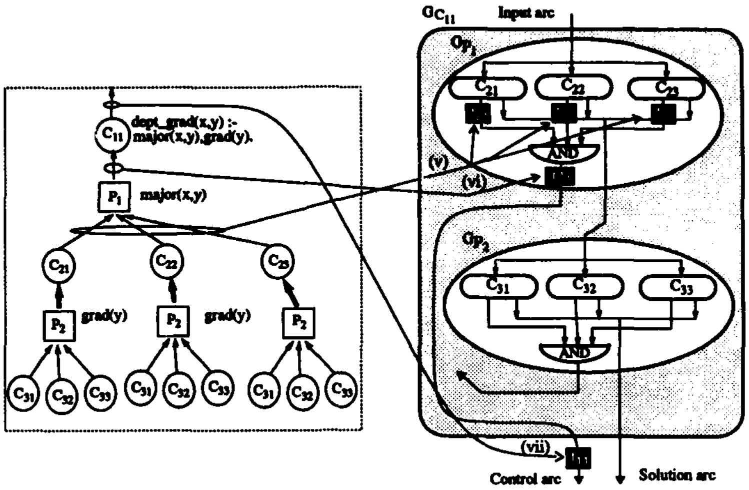


Figure 4: Example of backward propagation (2)

4.1 Structure of Program Execution

Fig. 5 depicts one possible state of program execution in the NDF model. The *activation contexts* are distributed over PEs during the data-driven execution outlined in Sec. 3. There exist two types of activation contexts: (1) Predicate activation contexts are primarily used for scheduling purposes such as backtracking and program completion. They comprise an argument frame, the information related to the scheduling of the participating clauses, and the continuation of the backtracking subgraph discussed in the previous section. (2) Clause activation contexts comprise a local environment frame as well as conditional bindings.

4.2 Unification and Generation Parallelism

The NDF model exploits unification parallelism only for the head literals whose unification always succeeds. They are determined through static analysis at compile time and appropriate annotations are inserted in the code generated by the compiler. However, the details are not presented here due to space limitations. Generation parallelism is always exploited without restriction. One additional slot, which is initialized with the number of arguments, is introduced in the argument frame for synchronization purposes. Instructions used in argument generation (e.g., "put-*xxx*" in the WAM) are extended so as to decrement the value and to signal the completion of argument generation when it becomes '0'.

4.3 Instantiation Parallelism

Instead of a stack, the heap memory in the NDF model is organized with frames managed with dynamic allocation and deallocation. The two operational modes of the "unify-*xxx*"

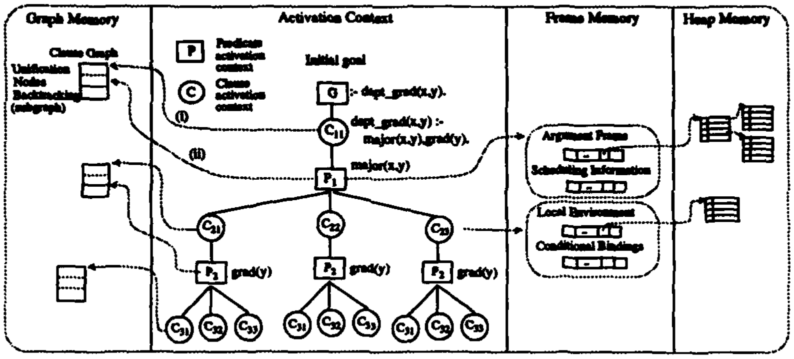


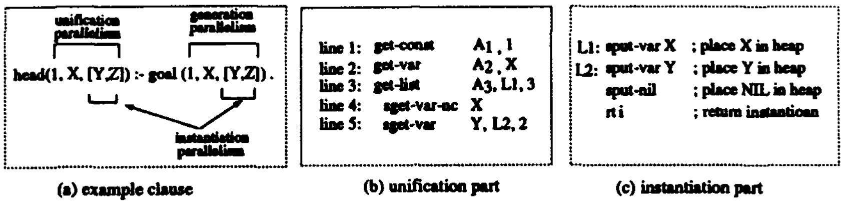
Figure 5: Basic structure of program execution in the NDF model

instructions in the WAM are separated into independent instructions for the purpose of making the instantiation parallelism in head unification explicit. They are "sget-*xxx*" and "sput-*xxx*", respectively for the read and the write mode. (The detailed description of the instructions is found in [6].) The compiled code for unification of a head literal is thus composed of two modules: one for pure unification and the other for instantiating complex terms in the head literal. The frame-based heap and explicit instantiation code on top of instructions designed for the *cdr*-coding, an optimized representation of list, render the maximal exploitation of instantiation parallelism feasible.

Fig. 6 shows a code for head unification of an example clause. As opposed to the WAM instructions, the "get-list" instruction has two additional operands, the starting address of the routine to make an instantiation of the corresponding complex term and the size of the term. The instruction allocates a frame with the size specified as the first operand and then calls the routine specified as the second operand, provided the dereferenced object of the argument is a variable. "sget-var" in line 5, that is to unify a variable inside a complex term, is also extended to have a format similar to "get-list" in line 3. Its operation has a similar role except the location and type of the term to unify with. The extension is to meet the need to instantiate any *cdr* part of a list in the head. For example, suppose that the argument term in A_3 is [3|Z], *i.e.*, the list consisting of "3" and "Z" as its *car* and *cdr*. After "get-list" at line 3, "sget-var-nc" at line 4 will bind "X" with 3. At line 5, "sget-var" instruction will instantiate a list [Y] by calling "L2", and then bind "Z" with its name.

5 Concluding Remarks

In this paper, we have described our self-scheduled inference mechanism and the corresponding fine-grain parallelism support in our NDF parallel execution model designed particularly for massively parallel architectures. As opposed to other data-driven approaches which generally consider only a tree expansion, the NDF graph embeds both


unification parallelism
generation parallelism
head(1, X, [Y,Z]) :- goal (1, X, [Y,Z]) .
instantiation parallelism

(a) example clause

```
line 1:  get-const      A1 , 1
line 2:  get-var        A2 , X
line 3:  get-list       A3, L1, 3
line 4:    sget-var-nc  X
line 5:    sget-var     Y, L2, 2
```

(b) unification part

```
L1: sput-var X   ; place X in heap
L2: sput-var Y   ; place Y in heap
    sput-nil     ; place NIL in heap
    rt i         ; return instantioan
```

(c) instantiation part

Figure 6: Compiled Codes for head unification

the tree expansion and retraction simultaneously into a data-flow graph. Thus, the NDF model is more complete in terms of execution control such as backtracking and allows distributed scheduling as a viable option. Further, as opposed to other models for distributed implementations, the capability to exploit fine-grain parallel by allowing multiple "active" tasks on a PE facilitates the efficient implementation on massively parallel architectures. In order to empirically verify the performance of the proposed parallel model, we have implemented a compiler for a pure logic kernel (*i.e.*, a subset) of PROLOG, together with an appropriate translator for the Fujitsu AP1000 distributed memory multiprocessor, and are now evaluating its performance.

References

[1] A. Ciepielewski, S. Haridi, and A. Hausman. OR-Parallel PROLOG on Shared Memory Multiprocessors. *Journal of Logic Programming*, 7:125–147, 1989.

[2] A. Gloria and P. Faraboschi. Instruction-level Parallelism in PROLOG: Analysis and Architectural Support. In *Proceedings of the 1992 International Symposium on Computer Architecture*, pages 224–233, May 1992.

[3] B. Hausman, A. Ciepielewski, and A. Calderwood. OR-parallel PROLOG Make Efficient on Shared Memory Multiprocessor. In *1987 Symposium on Logic Programming*, pages 69–79. IEEE Computer Society Press, August 1984.

[4] P. Kacsuk. *Execution Models of* PROLOG *for Parallel Computers.* The MIT Press, 1990.

[5] L. Kale. The Reduced-OR Process Model for Parallel Execution of Logic Programs. *Journal of Logic Programming*, 11:55–84, 1991.

[6] H. Kim and J-L. Gaudiot. The NDF Model: Processing Logic Programs on Large-Scale Parallel Architectures. Technical Report Number: CENG-94-10, EE-system, University of Southern California, January 1994.

[7] M. Rawling. GHC on the CSIRAC II Dataflow Computer. Technical Report TR-DB-91-05, Division of Information Technology, CSIRO, Australia, July 1991.

[8] D. Warren. Implementation of PROLOG - Compiling Predicate Logic Programs. Technical Report Vols. 1 and 2, Reports Nos. 39 and 40, Department of Artificial Intelligence, University of Edinburgh, May 1977.

PART VIII

APPLICATION SPECIFIC ARCHITECTURES

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

From SIGNAL to fine-grain parallel implementations

Olivier MAFFEÏS[a] and Paul LE GUERNIC[b]

[a]GMD I5 - SKS, Schloss Birlinghoven, 53754 Sankt Augustin, Germany

[b]IRISA, Campus de Beaulieu, 35042 Rennes CEDEX, France

Abstract: This paper introduces a new abstract program representation, namely Synchronous-Flow Dependence Graph (SFD Graph), which defines a generalization of DAGs and has been designed for the compilation of SIGNAL. In this paper, we emphasize the use of SFD Graphs for formal verification as well as implementation inference.

Keyword Codes: D.3.4, D.1.3, D.2.4,
Keywords: Programming Languages, Processors; Programming Techniques, Concurrent Programming; Software Engineering, Program Verification.

1 Introduction

In addition to the classical requirements of reliability and efficiency for the software, the spectacular technological advances in high performance computing has imposed a third one: portability. To cope with these three requirements, the programming languages community re-focused its attention in two complementary directions:

1. upstream, *language design to bring portability.* An outstanding work in this direction has been achieved from SISAL [5] which, although it is a fine-grain parallel language, has been implemented efficiently on various architectures [3].
2. downstream, *abstract representation design to gain reliability and efficiency.* For SISAL, this complementary approach yielded *IF1* [13].

Although the contribution of this paper aids both directions, it emphasizes the second one. Section 2 presents SIGNAL, a dataflow language based on the *(strong) synchronous approach* [6]; the specification of a variable illustrates the architectural independence of SIGNAL. In section 3, we present the main singularity of SFD graphs: the equational control model and how formal verifications over SIGNAL programs are performed with it. Section 4 actually defines the concept of SFD graphs. Finally, section 5 describes the inference of fine-grain parallel implementations from SFD graphs.

2 The SIGNAL Language

As SIGNAL is a dataflow-oriented language, it describes processes which communicate through sequences of (typed) values with an implicit timing: *signals.* For instance, a signal X denotes the sequence $(\mathbf{x}_t)_{t\in \mathbb{N}/\{0\}}$ of data indexed by time.

This work has been initiated at IRISA. It has been completed at RAL and GMD where it has been supported by an ERCIM (European Research Consortium for Informatics and Mathematics) fellowship.

Kernel of SIGNAL

The kernel of the SIGNAL language includes the operators on signals and the process operators. Four kinds of operators act on signals:

- **instantaneous functions** is a class of operators which encompasses all the usual functions (**and**, `<=`, `+`, `fft`, ...) extended to act on signals. Let f a symbol which denotes a n-ary function acting on signals and $[\![f]\!]$ the corresponding function acting on values, the SIGNAL process $\mathtt{Y} := f\{\mathtt{X1}, \ldots, \mathtt{XN}\}$ specifies that
$$\forall t \geq 1 \quad \mathtt{y}_t = [\![f]\!](\mathtt{x1}_t, \ldots, \mathtt{xn}_t)$$
In the specified behavior, one may notice that the value $\mathtt{y}_t$ carried by Y at instant t is equal to the function $[\![f]\!]$ applied to the values held by X1, ..., XN at the same instant. This matter of fact outcomes from a special specification approach: the *(strong) synchronous approach* (see [6] for an overview). In the dataflow synchronous approach, the execution of the operators is assumed of zero duration, only the logical precedence of values on a signal represents elapsing time. Therefore, firing waits and implicit queueing of data are suppressed at the specification level.

- **shift register** explicits the memorization of data; it enables the reference to a previous value of a signal. For instance, the process `Y := X $1` defines a basic process such that $\mathtt{y}_1 = v0$ and $\forall t > 1 \quad \mathtt{y}_t = \mathtt{x}_{t-1}$ where $v0$ denotes an initial and constant value associated with the declaration of Y. By contrast with the last two operatoes, the signals referred in instantaneous functions or in the shift register must be bound to the same time index, the same *clock*.

- the **selection** operator allows us to draw some data of a signal through some boolean *condition*. *The process* `Y := X when B` specifies that Y carries the same value than X each time X carries a data and B carries the value *true* (B must be a boolean signal). Otherwise, Y is absent, i.e. Y carries no value.

- the **merge** operator combines flows of data. The process `Y := X1 default X2` defines Y by merging the values carried by X1 and X2 and giving priority to X1's data when both signals are simultaneously present.

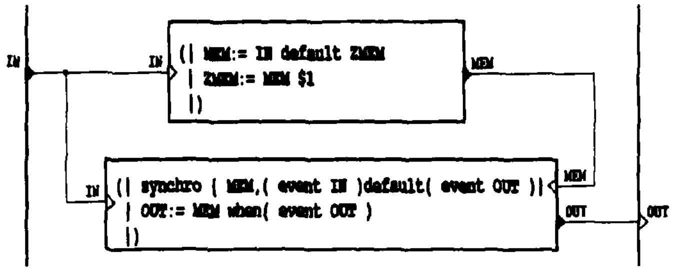
The four previous operators specify basic processes. The specification of complex processes is achieved with the parallel composition operator: the composition of two processes P1 and P2 is denoted (| P1 | P2 |). In the composed process, the common names between P1 and P2 refer to common signals; they stand for the communication links between P1 and P2. This parallel com· osition is an associative and commutative operator.

Specification of a memory cell

Externally, a variable is a device which memorizes the last written value —signal IN— and delivers it —signal OUT— when requested. The SIGNAL process VAR which specifies such a device is presented in Fig. 1; this figure is a screen-dump of the SIGNAL block-diagram editor.

The top box specifies the functional part of the memory: the content of the memory carried by the signal MEM is defined by IN when it occurs; otherwise, it keeps its previous value memorized by ZMEM. If ZMEM is initialized with zero, the sequence of values on IN, MEM and ZMEM may be:

IN	:	2			5	8		6			...
ZMEM	:	0	2	2	2	5	8	8	6	6	...
MEM	:	2	2	2	5	8	8	6	6	6	...



```
(| MEM:= IN default ZMEM
 | ZMEM:= MEM $1
 |)
```

```
(| synchro ( MEM,( event IN )default( event OUT ))
 | OUT:= MEM when( event OUT )
 |)
```

Fig. 1: Specification of the variable process **VAR**

This specification implies that the activity of memory device —the occurrences of **MEM**— is not restricted to its memorizing activity —the occurrences of **IN**.
The two basic processes included in the bottom box respectively specify:

1. *the control of the memory device.*

   The **event** operator makes explicit the transition from signals to *clock-signals which* are signals occurring only with the value *true*. If C is defined by `C:= event X` , which is rewritten in the kernel of SIGNAL by `C:= (X=X)`, a possible sequence of values may be:

   ```
   X : 2   5 3   8    6 ...
   C : t   t t   t    t ...
   ```

   The **synchro** operator specifies the synchrony of SIGNAL expressions. Therefore, the upper basic process of the bottom box in Fig. 1 states that **MEM** is defined when a write event occurs (**event IN**) or (**default**) an output is requested (**event OUT**).

2. *the definition of the output.* The memorized value carried by **MEM** is delivered in a *demand-driven* manner, when OUT is requested: **when(event OUT)**.

The specification of this memory cell emphasizes the architectural-independence of SIGNAL programming. This independence outcomes from the synchronous specification approach and the dataflow/equational style of the SIGNAL language. Therefore, the inference of reliable and efficient implementations is achieved in two steps. Firsly, we intend to *validate the specification independently from any target architectures* as sketched in the next two sections. Secondly, we intend to inference of efficient and semantics-preserving implementations as presented in section 5.

3 Control Validation & Inference

To comprehend the control model, let us recall the semantics of the selection operator. The process `Y := X when B` specifies that Y takes the value of X if X carries one (i.e., X occurs, X is present) and B is defined with *true*. Expressing the underlying control needs the status *present* and *absent* for X and, the status *present with true*, *present with false* and *absent* for B. The boolean algebra over $\{0, 1\}$ denoted by $\mathcal{B} =< C, \vee, \wedge, \hat{0}, \hat{1} >$, is taken to encode the status of the signals by:

present : 1   *absent* : 0

Let us denote $\hat{x}$ a characteristic function which encodes the states of X at each instant; $\hat{x}$ is called the *clock* of X; $\mathcal{B}$ is called the *clock algebra*

$[b], [\neg b]$ two characteristic functions or clocks which respectively encode the states *present with true* and *present with false* of the signal B

$\hat{0}$ the least element of $\mathcal{B}$ which stands for the never present clock; it is used to denote something that never happens

$\hat{1}$ the greatest element of $\mathcal{B}$, the always present clock

Among the above clocks, two equivalences hold: $[b] \vee [\neg b] = \hat{b}$ and $[b] \wedge [\neg b] = \hat{0}$. The first of these two equivalences means that, when a signal B carries a value, it is *true* or *false*. The second equivalence means that a boolean signal B cannot carry *true* and *false* at the same time. With this encoding, the of data implicit control relations of SIGNAL processes are encoded in equations over $\mathcal{B}$:

`Y:= f(X1, ..., XN)` :: $\hat{y} = \widehat{x1} = ... = \widehat{xn}$

`Y:= X $1` :: $\hat{y} = \hat{x}$

`Y:= X when B` :: $\hat{y} = \hat{x} \wedge [b]$

`Y:= X1 default X2` :: $\hat{y} = \widehat{x1} \vee \widehat{x2}$

With this clock encoding, the production/consumption of data in a complex program is translated into an intricate system of equations over $\mathcal{B}$. An equivalent representation is induced by projecting this equation system through the equivalence of clock. Beside the definition of an equivalent and compact representation of control, this projection may highlight control-flow inconsistency. For instance, the process

```
(| X:= A when (A>0)
 | Y:= X+A
 |)
```

is encoded by

$$\hat{x} = \hat{a} \wedge [a{>}0]$$
$$\hat{y} = \hat{x} = \hat{a}$$

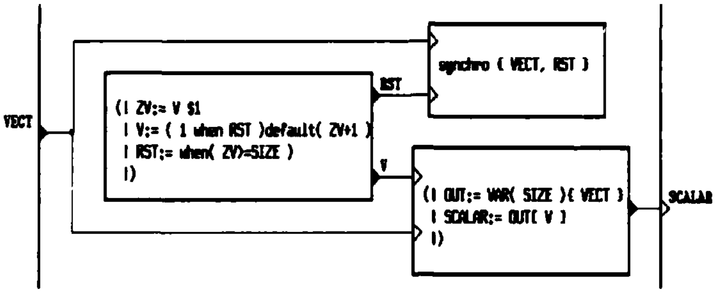
From this encoding, we deduce that $\hat{a} = \hat{a} \wedge [a{>}0]$. The solutions[1] of this equation stand for: there is no data on A ($\hat{a} = 0$), or A is carrying a positive value ($[a{>}0]{=}1$). In a classical dataflow interpretation of this program, if A holds a sequence of negative values, X carries no data, no firing of X+A is possible and the FIFO memorizing the values of A cannot be bounded. The above formal calculus exhibits such a possible dysfunction by excluding the case $[a{>}0]{=}0$ where an accumulation of data occurs. If non inconsistencies are detected, the FIFO queues used to buffer the data between any two operators in a dataflow interpretation can be bounded [4]. The reader interested in further details about control consistency over SIGNAL programs is referred to [9].

For further illustration purposes, let us consider the process UNFOLD specified in Fig. 2. This process converts a vector VECT —a spatial representation of data— into a sequence of SCALAR —a temporal representation of data. Such a process, which may be useful as front-end to achieve pipelined computations over VECT, has three components:

- left box: a *modulo counter*. Its current value held by V is reset to 1 when RST occurs; RST occurs when the counter reaches the limit defined by the parameter SIZE (a constant value). Note that `RST:= when Ψ` with `Ψ:= ZV >=SIZE` is the short form of `RST:= Ψ when Ψ`
- lower right box: an *enumeration process* of the SIZE elements of a vector signal OUT according to the occurrences of V. OUT memorizes the last occurrence of VECT by means of the process VAR specified in Fig. 1.

[1] since $([b] \vee [\neg b] = \hat{b}) \;\Rightarrow\; (\hat{b} = 0 \Rightarrow [b] = 0)$

- upper right box: a *control constraint* **synchro { VECT, RST }** which specifies that the enumeration of a vector must be completed before accepting another one.



```
(| ZV:= V $1
 | V:= ( 1 when RST )default( ZV+1 )
 | RST:= when( ZV>=SIZE )
 |)
```

```
synchro ( VECT, RST )
```

```
(| OUT:= WAR( SIZE )( VECT )
 | SCALAR:= OUT[ V ]
 |)
```

Fig. 2: Specification of UNFOLD

Applied to the example UNFOLD, the projection of the clock encoding highlights no flow inconsistency. From this projection, two classes of equivalent clocks are detected:

$$\{\widehat{\psi}, \widehat{v}, \widehat{zv}, \widehat{scalar}, \widehat{out}, \widehat{mem}, z\widehat{mem}\} \qquad \{\widehat{rst}, \widehat{vect}\}$$

These two equivalence classes, respectively identified by $\widehat{\psi}$ and $\widehat{rst}$, denote the signals which have equivalent rates in their flow of data. If Σ denotes the initial equation system over a set C of clocks, the projection of the control of UNFOLD is[2]:

$$C/_{\equiv} \;= \{\widehat{rst}, \widehat{\psi}\} \qquad\qquad \Sigma/_{\equiv} \;= \{\widehat{rst} = [\psi]\}$$

Mathematical properties of the control representation

The production/consumption of data in a SIGNAL process is represented by a system of clock equations over the clock algebra $\mathcal{B} = < C, \vee, \wedge, \widehat{0}, \widehat{1} >$ which is a particular boolean algebra. As boolean algebras are lattices, an alternative representation of $\mathcal{B}$ is achieved with a partial order:

$$< C, \leq > \qquad \text{with} \qquad \widehat{c} \leq \widehat{d} \iff \widehat{c} \vee \widehat{d} = \widehat{d} \quad (\Leftrightarrow \widehat{c} \wedge \widehat{d} = \widehat{c})$$

From the basic equivalences $[b] \vee [\neg b] = \widehat{b}$ and $[b] \wedge [\neg b] = \widehat{0}$, we deduce $\widehat{b} \wedge [b] = [b]$ and $\widehat{b} \wedge [\neg b] = [\neg b]$. Thus, among the basic clocks, the following two orders hold $[b] \leq \widehat{b}$ and $[\neg b] \leq \widehat{b}$. These two orders denote the property: a boolean signal carries a value more frequently than one particular (*true* or *false*) value. By using this second representation of the clock algebra, the equ^tion $\widehat{rst} = [\psi]$ induces: $\widehat{rst} \leq \widehat{\psi}$.

From this inequality, we infer that $\widehat{\psi}$ is the condition which controls the entire activity of the process UNFOLD. Beside the detection of control conditions used for implementations, it is this lattice form of the clock algebra which is handled by the heuristic algorithm (it is NP-complete problem) of SIGNAL compiler that projects a system of boolean equations through the equivalence of variables. By this projection, the SIGNAL compiler simultaneously (a) verifies the control consistency to ensure an execution over a bounded memory and (b) synthesizes a minimized control representation. In the next section, further validation of SIGNAL specifications is achieved by detecting deadlocks over SFD graphs.

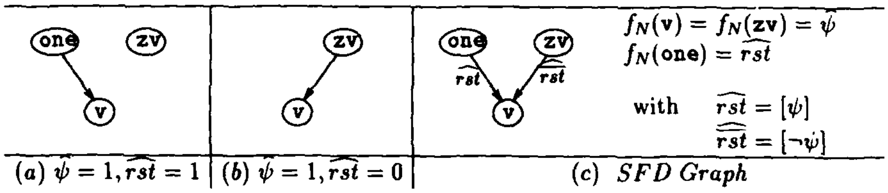
[2] The projection through $\equiv$ does not delete the equivalence $\widehat{rst} = [\psi]$ since $[\psi]$ represents some data selection over the signal Ψ; it may be seen as an "input" to the equational control model

4 Synchronous-Flow Dependence Graphs

To comprehend the requested representation of the data part of SIGNAL programs, let us consider the counting process of UNFOLD:

```
(| V:= ( 1 when RST )default( ZV+1 ) | ZV:= V $1 |)
```

As expressed by the control equations ($\widehat{rst} \leq \widehat{\psi}$), the control state where $\widehat{\psi} = 0$ and $\widehat{rst} = 1$ is unreachable. According to the and reachable and active[3] states of $\widehat{\psi}$ and $\widehat{rst}$, the different instantaneous (i.e, the dependency outcoming from the shift register is abstracted) data-dependence graphs that may occur are depicted in Fig. 3-a, Fig. 3-b.



$f_N(\mathbf{v}) = f_N(\mathbf{zv}) = \widehat{\psi}$
$f_N(\mathbf{one}) = \widehat{rst}$

with $\widehat{rst} = [\psi]$
$\widehat{\widehat{rst}} = [\neg\psi]$

$(a)\ \widehat{\psi} = 1, \widehat{rst} = 1$ | $(b)\ \widehat{\psi} = 1, \widehat{rst} = 0$ | (c) *SFD Graph*

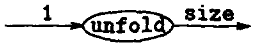
Fig. 3: Data-Dep. Graphs and the associated Synchronous-Flow Dependencc Graph

As $\widehat{\psi}$ identifies the equivalence class in which the clocks of V and ZV belong (cf p. 5), their nodes exist in Fig. 3-a and 3-b. When $\widehat{rst} = 1$, V is defined by ONE which is a signal that constantly carries the value 1; otherwise, the value of V is defined[4] by the value of ZV.

The abstract representation of the flow of data using Synchronous-Flow Dependence graphs (SFD graphs) is defined by superimposing all the possible instantaneous data-dependence graphs. Superimposing all the data-dependence graphs drawn in Fig. 3-a and Fig. 3-b induces the SFD graph depicted in Fig. 3-c. The paths taken by the data according to the control are described over SFD graphs by means of two mappings f_N and f_Γ: $f_N(\mathbf{one}) = \widehat{rst}$ means that ONE occurs when RST occurs ($\widehat{rst} = 1$) and $f_\Gamma(\mathbf{zv}, \mathbf{v}) = \widehat{\widehat{rst}}$ means that V is defined from ZV when RST does not occur. The new clock-label $\widehat{\widehat{rst}}$ denotes the control state when RST does not occur: $\widehat{\widehat{rst}} = 1$ when $\widehat{\psi} = 1$ and $\widehat{rst} = 0$ [5].

A SFD graph is defined by connecting a dependence graph to an equation system over clocks. A formal definition of SFD graphs is given in [12]. The SFD graph of the process UNFOLD is presented in Fig. 4; square nodes symbolize interface vertices; The nodes **mem** and **zmem** in this figure come from the instance of the process VAR.

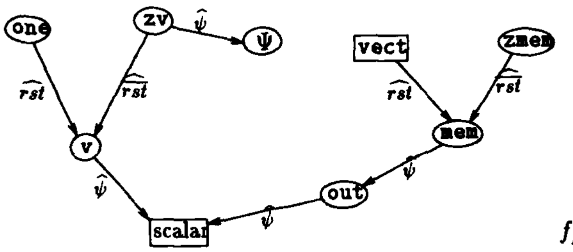
A similar and complementary approach to ours has been achieved in [10] over the concept of *Synchronous Data Flow Graph.* In this representation, the process UNFOLD which enumerates on its output the **size** elements of the input vector is represented by a node:



[3] The state $\widehat{\psi} = 0, \widehat{rst} = 0$ is ignored since no signal is present at this state, i.e nothing happens

[4] For presentation reasons, we have substituted $\mathbf{ZV} + 1$ by ZV.

[5] its definition is built over the negation of $\widehat{rst}$ which is denoted by $\widehat{1} \setminus \widehat{rst}$. Among the basic clocks, the following equivalences hold: $\widehat{b} \setminus [b] = [\neg b]$ and $\widehat{b} \setminus [\neg b] = [b]$. These equivalences respectively means that, when a boolean signal B carries a value but not *true*, it is *false* and the converse. Using these properties ($\widehat{\widehat{rst}} = \widehat{\psi} \wedge (\widehat{1} \setminus \widehat{rst}) = \widehat{\psi} \setminus [\psi] = [\neg\psi]$), the definition of $\widehat{\widehat{rst}}$ is simplified in $[\neg\psi]$.



$$\begin{aligned} C/_{\equiv} &= \{\widehat{rst}, \hat{\psi}\} \\ \Sigma/_{\equiv} &= \{\widehat{rst} = [\psi], \\ &\qquad \widehat{\overline{rst}} \equiv [\neg\psi]\} \\ f_N^{-1}(\hat{\psi}) &= \{\mathtt{v}, \mathtt{zv}, \mathtt{zmem}, \mathtt{mem} \\ &\qquad \Psi, \mathtt{scalar}, \mathtt{out}\} \\ f_N^{-1}(\widehat{rst}) &= \{\mathtt{one}, \mathtt{vect}\} \end{aligned}$$

Fig. 4 The Synchronous-Flow Dependence Graph of **UNFOLD**

Then, the consistency in the composition of foldings/unfoldings of vectors can be easily verified over Synchronous Data Flow Graphs. SFD graphs can also be compared are DAGs. A purely computational program is expressed in SIGNAL with exclusively instantaneous functions: all the signals have the same clock. Consequently, a single clock labels the associated SFD graph. In this case, we can ignored this (unique) labeling and *SFD graphs are equivalent to Directed Acyclic Graph (DAGs) [1].* Conversely, we assert that

> *SFD graphs define a generalization of DAGs to represent purely computational programs as well as programs with a complex* `if-then-else` *control structure*[6].

By contrast with DAGs, the clock labeling provides SFD graphs with a dynamical feature. This labeling imposes two constraints which are implicit for DAGs:

- *an edge cannot exist if one of its extremity nodes does not exist* is translated into the clock algebra, the image set of the mappings, by:
$$\forall(\mathtt{x},\mathtt{y}) \in \Gamma \qquad f_\Gamma(\mathtt{x},\mathtt{y}) \leq f_N(\mathtt{x}) \wedge f_N(\mathtt{y})$$
- *a cycle of dependencies stands for a deadlock* is verified over DAGs by definition; over SFD graphs, it is expressed as:

  a SFD graph $< G, C, \Sigma, f_N, f_\Gamma >$ *is deadlock free iff, for every cycle* $\mathtt{x1},\ldots,\mathtt{xn},\mathtt{x1}$ *in* G, $\quad f_\Gamma(\mathtt{x1},\mathtt{x2}) \wedge f_\Gamma(\mathtt{x2},\mathtt{x3}) \wedge \ldots \wedge f_\Gamma(\mathtt{xn},\mathtt{x1}) = \hat{0}$

  Intuitively, this equation translates the properties that a deadlock does not exist if all the dependencies of a cycle in a SFD graph are not present at the same time.

5 From SFD Graphs to Implementations

Reliability is the first requirement that an implementation must satisfy. From SIGNAL specifications, we intend to satisfy this requirement in two steps. Firstly, SIGNAL specifications are submitted to some formal checkers as sketched in the two previous sections. This reliability, thereby asserted at the specification level by this first step, is propagated further by means of the inference of semantics-preserving implementations.

Beyond reliability, implementations must run efficiently. Efficiency is reached by specializing specifications with respect to the target architectures. For SIGNAL, a first step towards efficient implementations is to give up successively with the two bases of SIGNAL's architecture independence: *its equational style* (cf sub-section 5.1) and *the null-time assumption of the synchronous approach* (cf sub-section 5.2).

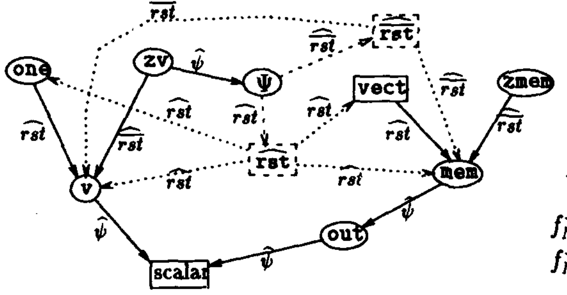
[6] we restrict the application field of SFD graphs to `if-then-else` controlled programs because no particular statement exists in Signal to specify iterations, they are only specified by means of temporal recursion using the shift register operator; the process **UNFOLD** illustrates such a specification.

5.1 Synchronous Implementation Inference

In the seek of an operational form of the control, we have considered two operational models: automata and dataflow graphs. We choose the dataflow option because of (a) the poor scalability and (b) the not natural adequateness for parallelism of auomata. This option can be expressed by extending the graph ; art of SFD graphs. By keeping the same representation , the detection of semantics changes like the introduction of deadlock are easily detected by re-using the previously developed tools.

Practically, the extension of the graph part of SFD graphs comes to introduce control nodes and dependencies. This extension is performed in two steps. Firstly, the mappings f_N and f_Γ are translated into dependencies expressing the control of the flows of data. For instance, as $f_N(\mathbf{vect}) = \widehat{rst}$ means that the existence of **vect** depends on $\widehat{rst}$, the activation dependency **rst** $\cdots\!\!\rightarrow$ **vect** is added. Also, if we assume that the control expressed by f_Γ is implemented in the target node, a dependency $\widehat{\widehat{\mathbf{rst}}}$ wil be added for the mapping $f_\Gamma(\mathbf{zmem}, \mathbf{mem}) = \widehat{\widehat{rst}}$.

Secondly, the way the added control nodes are computed is specified by orienting, from boolean selections to clocks, the equivalences enclosed in $\Sigma/_\equiv$. For instance, $\widehat{rst} = [\psi]$ is implemented by $\Psi \dashrightarrow \widehat{\mathbf{rst}}$. As clock-signal C is a signal occurring only with the value *true*, its clock is $\widehat{c}$: $f_N(\widehat{\mathtt{c}}) = \widehat{c}$. Finally, due to the inclusion property[7] that any SFD graph must satisfy,, the added dependencies between **x** and **y** are labeled with $f_N(\mathbf{x}) \wedge f_N(\mathbf{y})$. Proceeding this way, the synchronous implementation of UNFOLD which is induced is drawn in Fig. 5[8].



$$
\begin{aligned}
C/_\equiv &= \{\widehat{rst}, \widehat{\psi}\} \\
\Sigma/_\equiv &= \{\widehat{rst} = [\psi], \\
&\quad\ \widehat{\widehat{rst}} = [\neg\psi]\} \\
f_N^{-1}(\widehat{\psi}) &= \{\Psi, \mathbf{v}, \mathbf{zv}, \mathbf{zmem}, \\
&\quad\ \mathbf{mem}, \mathbf{scalar}, \mathbf{out}\} \\
f_N^{-1}(\widehat{rst}) &= \{\mathbf{one}, \mathbf{vect}, \widehat{\mathbf{rst}}\} \\
f_N^{-1}(\widehat{\widehat{rst}}) &= \{\widehat{\widehat{\mathbf{rst}}}\}
\end{aligned}
$$

Fig. 5: Synchronous implementation for UNFOLD

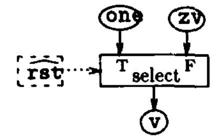
5.2 Asynchronous Implementation Inference

Indeterminism in asynchronous parallel languages is introduced through operators which possess a semantics taking into account the absence of data at a given time (e.g the guarded command in CSP). As execution times of these operators in an asynchronous approach depend on the execution support, the absence of data at a given instant is environment-dependent: the specified behavior of the whole system is environment-dependent. Owing to the synchronous approach which assumes the null-time duration of operators, the presence/absence of data is not dependent on the environment but is only application-dependent. Therefore, evenif SIGNAL's semantics refers to absence, the specified behaviors are deterministic.

[7] $\forall(\mathbf{x},\mathbf{y}) \in \Gamma \quad f_\Gamma(\mathbf{x},\mathbf{y}) \leq f_N(\mathbf{x}) \wedge f_N(\mathbf{y})$

[8] For readability reasons, the node $\widehat{\Psi}$ associated with the upper bound clock $\widehat{\psi}$ and its outgoing dependencies have been omitted.

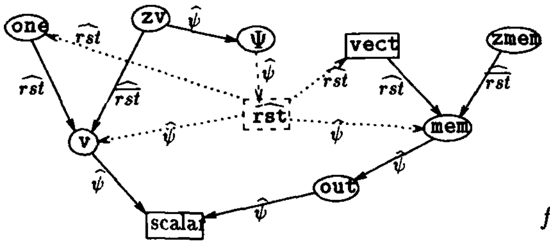
The implementation of SIGNAL's operators with asynchronous dataflow operators puts this null-time assumption into question. Consequently, preserving the determinism imposes a the control rewriting to ensure that the absence of data does not intervene. For instance, let us implement the process `V:= ( 1 when RST )default( ZV+1 )` by the dataflow graph



which imposes in terms of flow
$$f_\Gamma(\mathtt{one},\mathtt{v}) \vee f_\Gamma(\mathtt{zv},\mathtt{v}) = f_N(\mathtt{v})$$
$$f_\Gamma(\widehat{\mathtt{rst}},\mathtt{v}) = f_N(\mathtt{v})$$

The first line expresses a property over the flow of data; it is already verified by the **default** operator: no change is required. The second line expresses a property over the flow of control. It imposes that (a) $f_\Gamma(\widehat{\mathtt{rst}},\mathtt{v})$ must be equal to $\widehat{\psi}$ and (b) the control node $\widehat{\widehat{rst}}$ and its outgoing arcs become useless in this particular implementation.Consequently to this new labeling requirements, the activation of the control nodes must be rewritten to ensure the inclusion property over the resulting SFD graph. Formally, the condition that any control node c must verify is $f_N(\mathtt{c}) \geq \bigvee_{(\mathtt{c},\mathtt{v})\in\Gamma} f_\Gamma(\mathtt{c},\mathtt{v})$.

In the example UNFOLD, it is sufficient to set $f_N(\widehat{\mathtt{rst}})$ equal to $\widehat{\psi}$: $\widehat{\mathtt{rst}}$ becomes a boolean signal equivalent to Ψ. Finally, the SFD graph depicted in Fig. 6 specifies an asynchronous implementation of UNFOLD. The reader interested in a complete description of the rewriting process to infer asynchronous implementations is referred to [11].



$$C/_\equiv = \{\widehat{rst}, \widehat{\psi}\}$$
$$\Sigma/_\equiv = \{\widehat{rst} = [\psi], \; \widehat{\widehat{rst}} = [\neg\psi]\}$$

$$f_N^{-1}(\widehat{\psi}) \;=\; \{\Psi,\mathtt{v},\mathtt{zv},\mathtt{zmem},\widehat{\mathtt{rst}}, \mathtt{mem},\mathtt{scalar},\mathtt{out}\}$$
$$f_N^{-1}(\widehat{rst}) \;=\; \{\mathtt{one},\mathtt{vect}\}$$

Fig. 6: Asynchronous implementation of UNFOLD

6 Conclusion

The main contribution of this paper is to define a new abstract program representation called *Synchronous-Flow Dependence graph (SFD graph)*. This new abstract program representation has been designed to compile SIGNAL processes; SIGNAL is a synchronous dataflow-oriented language.

The major distinctive feature of SFD graphs is its equational control model which (a) defines an architecture-independent thus portable control representation and (b) eases the definition of tools which cope with applications with a complex control structure. SFD graphs are defined by connecting dependence graph to this equational control model; SFD graphs are a generalization of Directed Acyclic Graphs. This paper sketches tools for SFD graphs which perform:

- *Formal Verifications to ensure reliability.* The two formal tools presented in this paper enable: (a) to verify control flow consistency; it asserts that the execution can run over a bounded memory and (b) to detect deadlocks.

- *Inference of efficient implementations.* The inference of efficient implementations is performed by relaxing progressively the two architecture-independence bases of SIGNAL. Control calculus is required to ensure the inference of a semantics-preserving fine-grain parallel implementation.

SFD graphs and the tools presented in this paper are integrated in the SIGNAL compiler. This compiler is included in a CAD software environment which encompasses a graphic specification interface; a version of this CAD environment is commercially available.

Beside their utilization in the SIGNAL compiler, SFD graphs are used for the coupling with the SYNDEX system [8]: from SFD graphs, the SYNDEX system infers implementations onto various distributed architectures. In addition, SFD graphs are used as a common graph representation for the languages SIGNAL, LUSTRE [7] and ESTEREL [2]. More generally, SFD graphs constitute the conceptual basis of a proposal to the European Community for an Eureka Project which involves academic and industrial laboratories.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wiley, 1986.

[2] F. Boussinot and R. De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, Sept. 1991.

[3] D. C. Cann. Retire FORTRAN? a debate rekindled. In *Supercomputing 91*, pages 264–272. IEEE Press, Nov. 1991.

[4] P. Caspi. Clocks in dataflow languages. *Theoretical Comp. Science*, 94:125–140, 1992.

[5] J. Feo, D. Cann, and R. Oldehoeft. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, 10:349–365, Dec. 1991.

[6] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer, 1993.

[7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1321, Sept. 1991.

[8] C. Lavarenne, O. Segrouchni, Y. Sorel, and M. Sorine. The SYNDEX software environment for real-time distributed systems design and implementation. In *European Control Conference*, volume 2, pages 1684–1689, June 1991.

[9] M. Le Borgne, A. Benveniste, and P. Le Guernic. Dynamical systems over Galois fields and DEDS control problems. In *Proc. of the 30th IEEE conference on Decision and Control*, pages 1505–1510. IEEE Control System Society, 1992.

[10] E. A. Lee. Consistency in dataflow graphs. *IEEE Trans. on Parallel and Distributed Systems*, 2(2):223–235, April 1991.

[11] O. Maffeï. *Ordonnancements de graphes de flots synchrones; Application à* SIGNAL. PhD thesis, Université de Rennes 1, France, Jan. 1993.

[12] O. Maffeïs and P. Le Guernic. Combining dependability with architectural adaptability by means of the SIGNAL language. In *3rd Int. Workshop on Static Analysis*, pages 99–110. LNCS no 724, Springer-Verlag, Sept. 1993.

[13] S. Skedzielewski and J. Glauert. IF1: An intermediate form for applicative languages, version 1.0. Technical Report M-146, Lawrence Livermore National Lab., Mar. 1985.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Microcode Generation for Flexible Parallel Target Architectures

Rainer Leupers, Wolfgang Schenk, and Peter Marwedel

University of Dortmund, Department of Computer Science XII,
44221 Dortmund, Germany

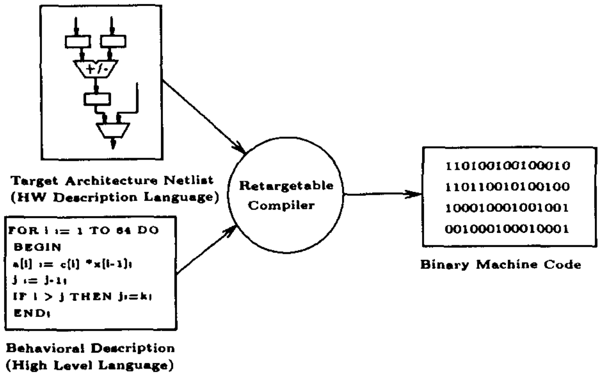
Abstract: Advanced architectural features of microprocessors like instruction level parallelism and pipelined functional hardware units require code generation techniques beyond the scoŗ ɜ of traditional compilers. Additionally, recent design styles in the area of digital signal processing pose a strong demand for retargetable compilation. This paper presents an approach to code generation based on netlist descriptions of the target processor. The basic features of the MSSQ microcode compiler are outlined, and novel techniques for handling complex hardware modules and multi-cycle operations are presented.[1]

Keyword Codes: B.1.4
Keywords: Control Structures and Microprogramming, Microprogram Design Aids

1 Introduction

Besides instruction pipelining, two important means for increasing the throughput of microprocessors have been identified by hardware designers: instruction level parallelism and data pipelining. Instruction level parallelism comprises several functional units working independently from each other, typically in combination with a VLIW type controller, whereas the latter is often used in digital signal processors (DSPs) for accelerating multiply-accumulate sequences. Exploiting these advanced architectural features poses new challenges for compiler technology, since there is no longer a clear compiler/architecture interface via an instruction set. Furthermore, recent processor design styles in the DSP area established a new view of the role of compilers in the design process. The use of ***application-specific instruction set processors*** (ASIPs) provides a convenient compromise between pure hardware implementations (ASICs) and pure software solutions via programmable off-the-shelf processors [1]. Usually, ASIP architectures are not fixed, but are subject to change during the design process. This implies "moving" pieces of functionality between hardware and software, which in turn requires frequent re-mapping of the system behavioral description onto the target architecture for performance evaluation. In order facilitate compilation onto different targets, the code generation process should be ***retargetable***, i.e. no manual compiler adaption should be necessary. We propose retargetability based on ***pure structural target descriptions*** at the register transfer level (fig. 1). The advantages of this approach are manifold:

[1] This work has been partially supported by ESPRIT BRA project 9138 (CHIPS)



```
FOR i := 1 TO 64 DO
 BEGIN
 a[i] := c[i] *x[i-1];
 j := j-1;
 IF i > j THEN j:=k;
 END;
```

```
110100100100010
110110010100100
100010001001001
001000100010001
```

Figure 1: Retargetable compilation based on structural descriptions

1) A RT level netlist of the target structure is usually available during the system design process.

2) Code generation is based on a model given in an easy-to-learn hardware description language. Thus, the concept "naturally" fits into a CAD system for design automation.

3) The controller structure is part of the architectural model, therefore restrictions due to encoding or sharing are detected by the compiler, and code generation is not restricted to VLIW type controllers.

4) With the controller structure being part of the model, it is possible to map resource conflicts to instruction conflicts, facilitating the scheduling phase.

5) Changes within the target architecture are easily reflected by adapting the architectural model.

6) No re-compilation of the compiler itself is required when moving to a new target architecture.

In this paper we present retargetable compilation techniques based on RT-level netlist models, which are capable of exploiting instruction level parallelism as well as data pipelining. Binary machine code is generated for *predefined* structures, in contrast to synthesis systems like CATHEDRAL [2] and CAPSYS [3], that perform binary code generation for *automatically synthesized* structures, which is a less expendable task. The CodeSyn compiler by Paulin [1] presents another approach to retargetable code generation for predefined structures which is based on specification of data flow patterns within the target hardware. However, these data flow patterns have still to be entered manually.

The paper is organized as follows. Sections 2 and 3 describe the modelling of architecture and behavior using the MIMOLA language. The basic steps for retargetable code generation (preallocation, pattern matching and scheduling/compaction) are presented in sections 4 to 6. These techniques have been implemented within the MSSQ compiler, which is part of the MIMOLA Design System [4]. Several restrictions of MSSQ have now been eliminated, e.g. pipelined modules and residual control are supported in the current version. These novel features are described in section 7. The paper ends with practical results and a conclusion.

2 Architectural models

The target architecture is modelled as a netlist on the register transfer level based on the MIMOLA language[2] with a PASCAL-like syntax. RTL modules are defined by their behavior based on a large set of primitive operations (arithmetic, logic, comparison, bit manipulation).

2.1 Combinational and sequential modules

Modules performing multiple operations are assumed to have a distinguished control input, e.g. a 16 bit ALU could be specified as follows, similar to a PASCAL procedure:

```
MODULE ALU (IN in1,in2:(15:0); OUT res:(15:0); FCT ctr:(1:0));
BEGIN
 res <- CASE ctr OF
         %00: in1 + in2;
         %01: in1 - in2;
         %10: in1 "AND" in2;
         %11: in1;
        END_CASE;
END;
```

Depending on the value of the control input `ctr`, the ALU either computes addition, subtraction or conjunction on the two data inputs `in1` and `in2` or passes `in1` to the output `res`. The data types are given as bitstrings in the format (`<highbit>:<lowbit>`). A 32 bit register with enabling signal and storing data at the rising clock edge is modelled by

```
MODULE Reg32bit(IN data:(31:0); OUT output:(31:0); FCT enable:(0); CLK clock:(0));
CONBEGIN
 CASE enable OF
  %0: "NOLOAD";                          (* do not load        *)
  %1: AT clock UP DO Reg32bit := data;   (* load at rising edge *)
 END_CASE;
output <- Reg32bit;                      (* read always        *)
CONEND;
```

The `CONBEGIN` ... `CONEND` construct denotes concurrent execution, in this case the register is always readable and concurrently stores input data at the rising edge when `enable = 1`. Memory modules are modelled similarly, having an additional address input. Modelling and code generation for multiport memories is provided.

2.2 Connections

Module interconnections are specified as a list of source and sink ports:

```
CONNECTIONS
  ALU.res         -> accumulator.input;
  accumulator.out -> ALU.in1;
```

[2] See [5] for the complete syntax. Convertors from VHDL to MIMOLA are available, but we prefer the latter throughout this paper for sake of better readability.

Bit subranges of modules ports may be referenced explicitly. Busses require a separate declaration due to their impact on the code generation process, i.e. the need for tristate operations of bus drivers:

```
BUS databus: (15:0);
```

2.3 Controller model

MSSQ was designed for code generation for microprogrammed structures. The underlying generic controller structure is depicted in fig. 2. One distinguished memory module has to be marked as the instruction memory and one register as the program counter. The next program address is determined by an arbitrary branch logic possibly dependent on several condition codes. Five versions of control flow are considered during code generation:

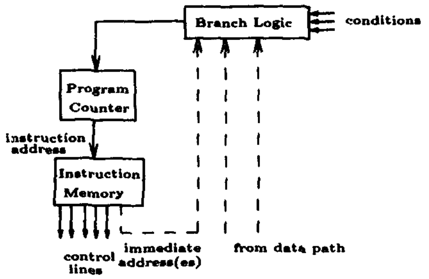


Figure 2: Generic controller structure

1) **increment program counter:** The program counter is set to the following program address.

2) **unconditional jump:** Continue at a certain program address.

3) **then-branch:** Branch if a condition is true, otherwise increment program counter.

4) **else-branch:** Branch if a condition is false, otherwise increment program counter.

5) **two-way branch:** Branch to address a_1 if a condition is true, otherwise branch to address a_2.

2.4 High-level transformations

Besides the netlist model comprising modules and interconnections, MIMOLA permits description of *replacement rules*, i.e. correctness-preserving transformations of operations. Such replacement rules allow compilation of operations that are not directly supported by the hardware. Possible replacements include

```
REPLACE &a * 2 WITH &a + &a END;
REPLACE &a * 4 WITH "SHIFTL"(&a,2) END;
```

where **&a** denotes a formal parameter of the replacement rule. Replacement rules may be unconditional (i.e. are always applied) or conditional (i.e. the compiler decides on demand whether or not to apply the rule). A set of standard rules, e.g. for replacing high-level language constructs like FOR or WHILE loops by conditional jumps are provided in a library. Other replacements may be specified by the user.

3 Behavioral models

MIMOLA is a unified language for describing both structure and behavior. Behavioral descriptions in MIMOLA are essentially PASCAL programs. Several deviations exist regarding the allowed data types. MIMOLA programs permit bit-level addressing, direct access to hardware storages and calling hardware modules like procedures. Therefore, behavioral descriptions can be specified at different levels of abstraction, for instance the following two programs are valid and equivalent:

```
PROGRAM AtHighLevel IS            PROGRAM AtRTLevel IS
TYPE Integer = (15:0);            BEGIN
VAR a,b,c: Integer;                DataRAM[0] := 3 * DataRAM[1];
BEGIN                              accu := DataRAM[0];
 a := 3 * b;                      END;
 c := a;
END;
```

In the latter, the variables a and b are assumed to be located at cells 0 and 1 of memory **DataRAM**, and variable c has been physically mapped to register **accu**.

4 Preprocessing and preallocation

Several preprocessing steps are applied to the behavioral description.

1) Abstract user variables are mapped to physical memory locations.
2) High-level control structures (WHILE, FOR, REPEAT, ...) are replaced by equivalent conditional jump constructs. Only IF-statements may remain as control structures.
3) Unconditional replacement rules are applied.
4) Different implementations of remaining IF-statements are considered. This feature permits **extension of basic blocks** and thereby higher degrees of freedom for the microcode compaction phase. See [6] for an exhaustive discussion of IF-statement implementation.

During the *preallocation phase*, the *Connection Operation Graph* (COG) is constructed that represents the hardware structure. Vertices correspond to modules, and edges represent interconnections. Semantical knowledge about module operators is exploited by performing several local transformations within the COG. This includes entering additional paths for commutative operations and *via* operations. Via operations can be used for propagating values from a module input to the output using neutrals. When an ALU, for instance, can perform addition on the inputs i_1 and i_2, it implicitly has a via operation on each of the two inputs by setting the other one to zero. Allocation of via operations provides higher flexibility for data routing during code generation.

Analyzing the COG yields a set of *assertions*, i.e. necessary control codes that force modules to perform certain operations. A partial COG for the example ALU of section 2.1 is shown in fig. 3, assertions are denoted by exclamation marks. Besides the COG, the result of the preallocation phase is a list of partial control word settings (*versions*), each able to force execution of a certain operation on a certain module. All different versions are kept in order to provide greater flexibility for the code generation phase.

5 Pattern matching and allocation

After the preprocessing phase, the behavioral description to be mapped onto the target hardware consists of RT-level assignments. Assignment allocation in MSSQ is based on

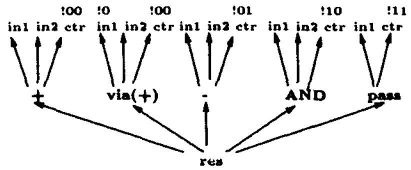


Figure 3: Partial Connection Operation Graph for an ALU

matching dataflow patterns with subgraphs of the COG.

5.1 Allocation of simple assignments

Considering the assignment accu := DataRAM[0] + 17 the following subtasks have to be performed:

1) Enable accu for loading data
2) Provide address 0 at DataRAM
3) If necessary, set DataRAM to a readable mode
4) Allocate the constant 17
5) Allocate the addition operation on an ALU
6) Route the operands DataRAM[0] and 17 through the circuit to the ALU inputs
7) Route the result to the target accu

The COG is traversed in order to find the required operators, in this case addition. Providing the necessary control codes and constants relies on the results of the preallocation phase. Data routing is based on the COG interconnect structure and may require additional control codes, e.g. when exploiting via operations. If all subtasks can be solved, the assignment is finally tranformed into a set of partial control word settings, concurrently necessary to execute the assignment. If allocation fails, MSSQ generates an error message indicating the failure reasons, e.g. missing operators or data routes.

5.2 Sequentialisation

Assignments containing complex expressions like

```
regfile[2] := (DataRAM[1] * accu) + (regfile[1] SHIFTL 2);
```

in general require to be sequentialised. In this case, MSSQ relies on a list of distinguished possible temporary locations specified in the MIMOLA input and computes possible sequential versions. The resulting sequential assignments are treated as simple assignments.

5.3 Conditional assignments

After replacing high-level control structures in the preprocessing phase, assignments still may contain IF-statements, e.g.

```
IF cond THEN accu := DataRAM[0];
```

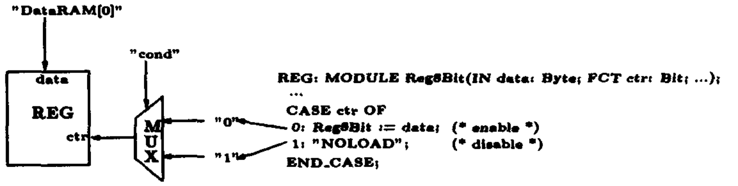
for which various implementations exist. Currently, two versions are implemented in MSSQ:

Conditional jump versions The above example can be transformed into the sequence

```
         IF cond THEN PC := PC + 1 ELSE PC := <label>
         accu := DataRAM[0];
<label>: <next instruction>
```

only requiring a multiplexer at the PC input. The assignment can be treated as a simple assignment.

Conditional load versions require a hardware structure as depicted in fig. 4, which often occurs in real microprocessors. Depending on a condition bit, the target storage module is either enabled or disabled. The data to be conditionally loaded is unconditionally routed to the storage data input, and the statement can remain unchanged. MSSQ tries to allocate both implementation versions. The better alternative is selected during microcode compaction.



```
REG: MODULE Reg8Bit(IN data: Byte; FCT ctr: Bit; ...);
 ...
 CASE ctr OF
 0: Reg8Bit := data;  (* enable *)
 1: "NOLOAD";         (* disable *)
 END_CASE;
```

Figure 4: Hardware for conditional load operations

6 Scheduling and compaction

The MSSQ scheduler heuristically tries to pack as many microoperations as possible into one machine instruction within each basic block. Since **resource conflicts are mapped to instruction conflicts**, it is sufficient to check whether the corresponding partial instructions are bit-compatible. Since all versions for execution of each statement are kept during the allocation phase, in case of incompatible operations the scheduler may select from several alternatives, and the shortest possible instruction sequence for each basic block can be selected. In addition to scheduling allocated assignments, any compiler based on structural hardware descriptions rather than on instruction sets has to **prevent undesired side effects** for each machine cycle. Two sources of side-effects must be taken into account:

Unused storage modules have to be disabled within each microinstruction in order to preserve their current state.

Unused bus drivers have to be disconnected by allocating tristate modes in order to avoid bus conflicts.

In general, additional control codes have to be supplied to prevent these side-effects. If a side-effect cannot be prevented, e.g. due to a missing register enable line, compaction fails and an error message is generated. The final result of the scheduling/compaction phase is a binary microprogram which realizes the specified behavior on the given target architecture.

7 Extensions for complex modules

MSSQ lacks from generating code for hardware structures with complex modules, such as pipelined modules, multiple cycle operations or multiple output operators. There are also restrictions imposed by the book-keeping of temporaries, which prevent the support of residually controlled modules.

We have overcome the deficiencies by developing a new approach, that accounts for complex modules, and has an improved data routing and book-keeping mechanism.

7.1 Complex module classification

There is a number of complex modules present in contemporary processors. They can be classified for code generation purposes – depending on the required control scheme – as described in the following.

Multiple cycle operations with fixed control are usually present at slow operators whose delay exceeds the cycle time. The code generator has to provide the control code stable during all cycles. The assumption, that each operation yields the result after an a priori known fixed number of cycles (the delay) is made.

Multiple cycle operations with initial control require the control code in the first cycle of an operation only. The operation takes multiple cycles to complete but needs no further control to do so.

Multiple cycle operations with variant control occur at programmable modules. The desired operation is decomposed into a sequence of more basic operations, each of which with a specific control code. The code generator emits code for each basic operation.

7.2 Residual control

A register is connected to a control input of the module. The code generator is therefore forced to load the desired control code into the register before the operation can be carried out. Loading the control code may be done one or more cycles in advance, but it must not be destroyed by an intermediate instruction.

7.3 Modeling complex modules

Whereas a key feature in the approach developed with MSSQ is the modeling of resource conflicts as instruction conflicts, the new approach deals with a redefined notion of resources, that suits the need for tracking the hardware resource usages over an interval of cycles. A *resource* is a register, a memory cell, a signal or an instruction field. A resource may be occupied by a value in a specific number of cycles. A *resource usage* is represented as a triple (r, v, i), where r is a resource, v is a value, and i denotes an interval of cycles. The following pipelined ALU latches all inputs in each cycle. It is of the initial control type.

```
MODULE MulDiv(IN a,b,c: int; OUT o: long);
VAR la,lb,lc: int;
CONBEGIN
 la := a; lb := b; lc := c;
 CASE lc OF
  0: o <- la * lb;
  1: o <- la / lb;
 END
CONEND;
```

The resources of the module are the signals a,b,c,o and the latches la,lb,lc. The latches are unconditionally loaded within each cycle. Such carriers are called ***pipeline registers***. The behavior analysis extracts sets of resource usages for each path from a data source to a data sink. A path may include several pipeline registers, since pipeline registers are not regarded as data sink. The book-keeping mechanism keeps track of the variable ***bindings*** as well as the ***machine state***. The machine state is a mapping of the resources to the values. Variables carried at a resource are said to be bound to it.

7.4 Code generation

A set of resource usages may contain one or more outputs of an operation, a set of side effects of the operation and the set of prerequisites (or assertions). A ***template*** is the partition of a set of resource usages into A, S and R. The elements of A, S, R are called *assertions*, *side effects* and *results* respectively. An operation is allocated when all prerequisites are allocated. If the resource denotes an instruction field, the value is a partial code version for the operation. If the resource refers to a register or storage cell, the book-keeping mechanism is considered whether or not the required value is already present. The set of side effects is used to update the book-keeping of the current machine state. Allocation, data routing and compaction can influence each other. During allocation of a statement, the allocator collects partial code versions. The code generator checks for resource conflicts. Two resource usages $u_1 = (r_1, v_1, i_1), u_2 = (r_2, v_2, i_2)$ are *conflicting*, if $r_1 = r_2$ and $v_1 \neq v_2$ and $i_1 \cap i_2 \neq \emptyset$.

Whenever a (non pipeline) register has to be used as a temporary, the resulting partial code is tentatively compacted. If there is no valid schedule, because resource contention is exposed, a backtracking step is initiated. This scheme results in an exhaustive search for data routes.

With these extensions we expect a more versatile tool for a broad range of target architectures. The backtracking approach in allocation, data routing and compaction explores all versions for implementing a given basic block. Thus we can trade off the quality of the generated code against the time spent in searching for alternative versions.

8 Results

The MSSQ microcode compiler has been implemented by a total of about 34,000 PASCAL code lines and has been applied to a variety of real-life designs. These include:

- Verification of the SAMP processor [7].
- Code generation for the PROLOG processor PRIPS, which was recently fabricated through EUROCHIP.

- Assembly code generation for TI's TMS320C25 DSP, based on a novel code generation methodology [8].

Typical compilation rates are between 10 and 100 instructions per second on a SUN SparcStation 10. This is acceptable for the intended application area, i.e. code generation for flexible target architectures instead of standard processors.

9 Conclusions

A feasible approach to code generation for flexible programmable target architectures was presented. Due to higher compilation times, retargetable compilers are not expected to replace target-specific compilers for standard processors. The main application area can be identified as code generation for ASIPs. According to Paulin [1], there is a clear trend towards ASIPs as a design style for DSP systems. Compiler retargetability facilitates the selection of an appropriate ASIP architecture that meets the given timing constraints. In addition, the trade-off between hardware and software implementations of particular functions is supported.

References

[1] C.Liem, T.C.May, P.G.Paulin: Instruction Set Matching and Selection for DSP and ASIP Code Generation, Proc. European Design and Test Conference (EDAC), 1994

[2] D. Lanneer, F. Catthoor, G. Goossens, et al.: Open-ended System for High-Level Synthesis of Flexible Signal Processors, Proc. European Conference on Design Automation (EDAC), 1990, pp. 272-276

[3] G. Menez, M. Auguin, F Boeri, C. Carriere: Contribution of Compilation Techniques to the Synthesis of Dedicated VLIW Architectures, IFIP Transactions on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism (A-23), 1993, pp. 217-228

[4] P. Marwedel, W. Schenk: Cooperation of Synthesis, Retargetable Code Generation and Test Generation in the MIMOLA Software System, European Design and Test Conference (EDAC), 1993, pp. 63-69

[5] R. Jöhnk, P. Marwedel: MIMOLA Reference Manual V 3.45, Technical Report No. 470, available from: University of Dortmund, Dept. of Computer Science, 44221 Dortmund, Germany

[6] P. Marwedel: Implementation of IF-statements in the TODOS microarchitecture synthesis system, IFIP Trans. on Synthesis for Control Dominated Circuits (A-22), 1993, pp. 249-262

[7] L. Nowak: SAMP: A General Purpose Processor Based on a Self-Timed VLIW Structure, ACM Comp. Arch. News, Vol. 15, No. 4, 1987, pp. 32-39

[8] R. Leupers, W. Schenk, P.Marwedel: Retargetable Assembly Code Generation by Bootstrapping, Proc. 7th International Symposium on High Level Synthesis, 1994

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)
© 1994 IFIP. All rights reserved.

A Fleng Compiler for PIE64

Hidemoto NAKADA, Takuya ARAKI, Hanpei KOIKE, Hidehiko TANAKA

H.Tanaka Lab., Dept. of Electrical Engineering, Faculty of Engineering,
University of Tokyo
7-3-1, Hongo, Bunkyou-ku, Tokyo 113, Japan
Email: {nakada,araki,koike,tanaka}@mtl.t.u-tokyo.ac.jp

Abstract: The programming language fleng is designed for highly-parallel execution of non-uniform problems. We cannot expect data concurrency from these problems. Therefore, fleng makes use of control concurrency which is derived from data dependency. Fleng execution efficiency depends on load distribution and scheduling.

We implemented a fleng system on the parallel inference engine PIE64. Our compiler performs static load partitioning and scheduling. Static load partitioning improves concurrency and reduces remote memory references. Static scheduling reduces synchronization costs arranging process execution order. These static optimizations require data-flow analysis. In this paper, we describe a data-flow analysis method in our system.

Keyword Codes: D.3.m
Keywords: Programming Languages, Miscellaneous

1 Introduction

In general-purpose highly-parallel systems, it is required to execute not only uniform problems, but also non-uniform problems, such as symbol processing. There has been many work for the uniform problems: for example, vectorizing and parallelizing compilers are available these days. These systems make use of data concurrency; however we cannot expect a lot of data concurrency from non-uniform problems. Therefore, these known techniques are inapplicable to non-uniform problems.

Fleng is a committed-choice language which can extract control concurrency from any problems. So, it is suitable for the highly parallel execution of non-uniform computation.

Efficiency of fleng execution depends on good methods for load distribution and scheduling. We describe our fleng compiler system for the parallel inference engine PIE64 focusing on static load partitioning and scheduling.

Load distribution itself is not difficult. However, naive distribution causes a lot of remote memory references and prevents efficient execution. The purpose of static load partitioning is to improve memory reference locality as long as maximum concurrency is maintained.

Hidaka [2] described static load partitioning based on execution profile. However, that method had two shortcomings: (1) it required too much time for analysis, and (2) it was restricted by the initial input for the profiler. Here we show a method based on data-flow analysis, which separates goals into several units along with the data flow.

The purpose of the static scheduling is to decrease the number of synchronization called suspension. Suspension arises from inappropriate execution order of goals. Therefore, suspension can be avoided to a certain extent by rearranging the order. We arrange goals according to a result of data-flow analysis.

Fleng and PIE64 are briefly described in section 2. Section 3 gives an overview of the whole system. The compiler and its static load partitioning and scheduling methods are described in section 4. Finally, we describe the current status and future plans in section 5.

2 Fleng and PIE64

2.1 Committed-Choice Language fleng

Fleng is a member of a logic-based language family called Committed-Choice Languages. This family is a descendant of concurrent logic languages[3]: GHC and KL1[1] are well known members of this family.

A fleng program is a set of predicates. A predicate consists of a set of clauses. Clause is the minimum fragments of the program. A clause consists of a **head** and **bodies**. We separate head and bodies by ":-".

$$Head : -Body1, Body2, Body3.$$

Basically, a clause is a description of a rewriting rule, and the head represents the pattern which the clause can rewrite.

One of the most notable features of fleng is a single assignment variable. Variables have only two states: at creation time variables are **unbound** and by **binding** they change to a second state called **bound**. Binding is similar to assignment, but once bound to one datum, the variables never revert to unbound, and cannot be bound to other datum. A second and any subsequent binding attempts will have no effect.

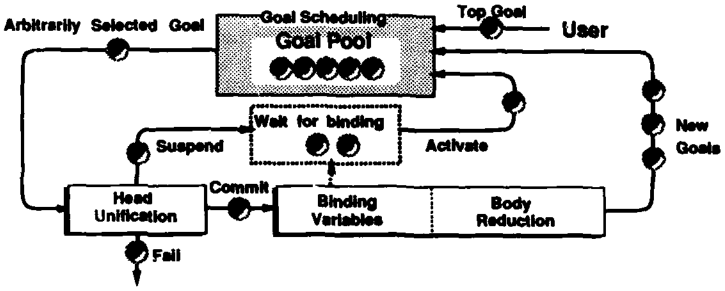


Figure 1: fleng execution model

Figure 1 shows a diagram of fleng execution. The basic execution unit of fleng is called a **goal**. While a fleng program is executed, there are many goals in the goal pool. Roughly speaking, the number of the goals in the pool is regarded as a measure of parallelism at the time. Fleng execution can be decomposed into three steps; (1) arbitrarily select a goal from goal pool, and then, (2) rewrite the goal to other goals, at last, (3) return new goals to the goal pool. The rewriting is called **reduction**, and is done by the rule described by a clause which has the same name as the goal. A user starts execution by placing a goal, called the **top goal**, into the goal pool. When the goal pool becomes empty, execution stops.

To rewrite one goal, it is necessary that arguments of the goal are sufficiently bound to match with some of the clauses. If this condition is not met, it is impossible to rewrite the goal at that time. To wait for the appropriate condition, the fleng system **suspends** the goal according to the variable which is needed for the condition. This means that the goal is removed from the goal pool and, until the variable is bound, the goal will never try to be rewritten again. When the variable is bound, the system puts the goal into the goal pool again. This operation is called **activation**. Because of this synchronization system, no matter how we select a goal or when we rewrite it, it is guaranteed that result of a program execution is not affected by goal execution order.

In fleng, synchronization is expressed as a binding to variable and pattern match in clause head. This implicit synchronization enables fine-grained highly-parallel execution.

2.2 Platform for fleng : PIE 64

PIE64 is designed for fleng, and has dedicated hardware to reduce the costs for communication and synchronization, and to support load balancing. PIE64 also has hardware which supports parallel management.

Overview of PIE64 PIE64 consists of 64 processing elements called IU(Inference Unit) and two interconnection networks. The most notable feature of each processing element of PIE64 is to have three kind of processors. We divide parallel execution into three parts; computation, communication/synchronization, and parallel management, and assign three kind of processors these roles.

To reduce the cost of remote communication, PIE64 adopts latency-oriented interconnection networks, dedicated communication processors, and computing processors with a multi-context facility to hide latency. The communication processors also have a facility to support synchronization directly, to reduce the cost of synchronization. To support the load distribution, the interconnection network has a facility called automatic load balancing. With this facility, all IU can have access to the least load level, and the least loaded IU.

IU: Inference Unit An IU has three kinds of tightly connected processors: UNIRED (Unifier / Reducer) for fleng execution, NIP (Network Interface Processor) for communication and synchronization, and MP(Management Processor) for management.

UNIRED is a dedicated processor to fine-grained symbol processing[4]. It is designed with tag architecture and a dedicated instruction set. It also has several special features for organizing of parallel machines. The pipeline of UNIRED is shared by four contexts to hide remote reference latency.

NIP has two roles; communication and synchronization. The memory of PIE64 is distributed in each IU, but has one global address space. NIPs organize these distributed memory into a distributed shared memory. NIP also supports synchronization using single-assignment variable. Synchronization among IUs has to include some communication, so it is inevitable that one processor is in charge of these two roles.

MP is dedicated to parallel management. We use a general purpose SPARC processor, as MP.

3 Fleng system on PIE64

In this section, we review requirements of the system, and then show a sketch of our system.

3.1 Load Distribution and Scheduling

Load distribution and scheduling are important problems for implementing highly-parallel language systems. The optimum solution to these problems depends on the run-time situation. However, it is very difficult to predict perfectly the run-time situation of a non-uniform application. Therefore, it is probably impossible to obtain the optimum load distribution and scheduling using only a compiler.

Load Distribution Requirements for load distribution are as follows:

- To extract concurrency,
- To balance load,
- To reduce communication.

First, sufficiently high concurrency has to be extracted to fill all the IUs with some goals. Second, load, i.e. goals and data, have to be balanced among IUs in order to avoid overload on particular IUs. Third, the goals and data have to be allocated in such a way as to reduce the communication between IUs.

Scheduling An important requirement for scheduling is to minimize synchronization cost. Synchronization cost in PIE64 is low, because PIE64 hardware supports synchronization. However, some cost still remains. Suspension causes context changes on UNIRED and MP, and their costs also cannot be neglected.

Suspension arises because of inappropriate execution order of goals. Assume that two goals, a producer and a consumer, share one variable. The producer will bind some value to the variable, and the consumer requires the value for reduction. If the consumer is scheduled before the producer, the consumer suspends. However, if the producer is scheduled first, neither of them suspends. Therefore, scheduling by considering data dependency reduces suspension.

3.2 Overview of fleng system

To cope with load distribution and scheduling, we have adopted combination of static optimization and dynamic control.

The fleng system for PIE64 consists of a compiler system and a run-time system. On PIE64, compiled code on UNIREDs and a **run-time kernel** on MPs cooperate to execute fleng programs.

Compiled code on UNIRED can concentrate on computation, since MP and NIPs take charge of other bothersome tasks. UNIRED only receives a goal from MP, and reduces it, and sends back new goals to MP. If a goal cannot be reduced, UNIRED simply quits execution. MP and NIPs are responsible for suspending the goal. Whenever remote access is required, UNIRED automatically sends a command to NIP.

The principal role of the run-time kernel is goal management. The goal management is to fill up all UNIREDs with goals. The goal pool is distributed across all IUs in order to avoid a bottle neck. Each MP manages a goal pool on the IU, i.e. it supplies UNIRED goal, gets new goals from UNIRED, and when the need arises, sends goals to other IUs or recieves goals from other IUs.

Load distribution on PIE64 Load distribution on PIE64 is handled at three stages as shown below.

1 Static load partitioning by compiler,
2 Dynamic load distribution by run-time kernel,
3 Dynamic load balancing by interconnection networks.

In the first stage, the compiler partitions loads; i.e. newly created goals and data. The role of the first stage is to enhance memory reference locality as long as all possible concurrency is maintained. The compiler detects data dependency between goals to allocate related goals to the same IU. In the next section, we will describe the details of this partitioning.

In the second stage, using run-time information, the run-time kernel decides whether the load should be distributed. Under a situation that all other IUs are sufficiently loaded, distributing load is not only useless, but can worsen reference locality. Therefore, under such a situation, the run-time kernel keeps all newly created goals inside the IU to suppress excessive concurrency.

At last, in the third stage, the automatic load balancing facility of interconnection networks is used to send goals to the least loaded IU.

4 Compiler and Static Optimization

4.1 Overview of Compiler System

Figure 2 shows a brief diagram of the fleng compiler system. This system consists of a mode analyzer, an optimizer and the compiler. All of them are written in fleng themselves.

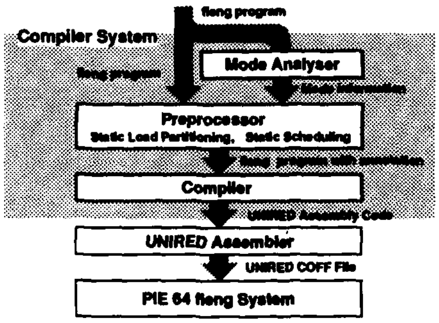


Figure 2: fleng compiler system

The optimizer does load partitioning and scheduling according to information obtained by the mode analyzer. Decisions made by the optimizer are added to the program as annotations. The compiler generates UNIRED assembly code according to the annotations. We separate the optimizer and the compiler to evaluate various kinds of optimization policy without any modification to the compiler.

4.2 Annotations

We use annotations to inform the compiler about the decisions determined by the optimizer.

The syntax of the annotation produced by the optimizer is as follows, where *Item* is a variable, a structured term or a body goal.

Item `@[` *annotation1, annotation2,....* `]`

The result of load partitioning is specified with the following annotation:

- `local`
  This annotation means that the goal/data should be allocated in the local IU.
- `on(` *label* `)`
  This annotation is used in the head part of clauses. The IU where the data resides is tagged as *label.*
- `to(` *label* `)`
  This annotation means that the goal/data should be allocated in the IU tagged as *label.*
- `any(` *label* `)`
  This annotation means that the goal/data should be allocated in the least loaded IU. The IU where the goal/data is allocated is tagged as *label.*

The result of static scheduling is specified with the following annotation:

- `sequence(` *number* `)`
  This annotation specifies an execution order of the goals which are allocated in the same IU.

4.3 Static load partitioning and Static scheduling

As mentioned in section 3, the role of static load partitioning is to enhance memory reference locality as long as all concurrency is maintained. To achieve this objective, data dependency information is required.

The point of static scheduling is to arrange goal execution order. This also requires data dependency information. In both cases, we get the information by data-flow analysis.

4.3.1 Simple mode analysis

Our data-flow analysis consists of two phases; mode analysis and data-dependency analysis. In the first phase we determine the mode of predicates. In the second phase, using the mode information, we analyze the data dependency in each clause.

Although there are several work on mode system for the committed-choice languages, they specify only input/output mode. The second phase requires data-dependency information, we create a new mode system which includes data-dependency information.

Predicate mode is represented by a list of argument modes. Our mode system uses the following 5 argument modes.

++ **strong input**: required immediately
+ **input**: required but not immediately
−− **strong output**: guaranteed to be bound
− **output**: not guaranteed to be bound
? **unknown**

For example, consider the following program fragment:

```
foo(a, B):- B = b.
```

This predicate requires that the first argument is bound, and guarantees that the second argument is bound after the execution. As a result, the mode of this predicate is specified as `[++, --]`.

To get full mode information, a global analysis is required. However, local analysis will be enough to get the data-dependency information. Therefore, we decided not to do global analysis.

The mode of predicates is determined by synthesizing mode of clauses. Clause mode is also specified as a list of argument modes. The mode of clause is determined by the mode of each arguments. The argument mode is determined by the following rules:

- In the head, if the argument is not a variable, the argument mode is specified as strong input.
- In the head, if the argument is a variable, and there are some goals which binds the variable to some non variable object, then the argument mode is specified as strong output.

The actual rules are somewhat more complicated, but omitted here.

The n-th argument mode of a predicate is synthesized from the n-th argument modes of clauses. Each argument mode is determined by the following rules:

- If all the modes of clauses are same, the mode is taken as the argument mode.
- If strong input and input conflict, input is selected.
- If strong output and output conflict, output is selected.
- If unknown and any other mode conflict, other mode is selected.
- If input and output conflict, unknown is selected.

4.3.2 Data-flow graph

From the predicate mode, we can get a data flow directed graph, using variables and goals as nodes, and data dependency as arcs. The strong-input mode indicates that a goal depends on an argument which have the mode. The strong-output mode indicates that an argument which have the mode depends on a goal. A data-flow graph can be obtained by the following steps:

1. Treat variables as data nodes.
2. Treat body goals as goal nodes.
3. If a body goal has a strong input mode on a variable, make arcs from the data node to the goal node.
4. If a body goal has a strong output mode on a variable, put arcs from the goal node to the data node which represent the variable.

4.3.3 Load partitioning and scheduling

Load partitioning and scheduling problems can be resolved into the data-flow graphs partitioning. For any walk in the graph, any two goals on the walk can not be reduced simultaneously. In other word, for any two goals, no graph walk includes both of them, they can be reduced simultaneously. Therefore, the load-partitioning problem can be solved by partitioning of data-flow graph.

The scheduling problem can also be solved by graph partitioning. In the data-flow graph, upstream goal should be reduced in advance. Therefore scheduling can be done by arranging the goals according to the graph walk.

We partition the data-flow graph with the following steps:

1. Select one of the longest walks as the target arbitrarily.
2. Remove all the goal/data nodes and arcs which are included in the target walk, and allocate them to one processor;
3. By the previous operation, if some data nodes are isolated from the graph, allocate the data node and arc to the same processor as 2;
4. By operation 2, If some goal nodes are isolated, allocate them on another processor;
5. Repeat these operation until the graph becomes empty. If the graph is separated into plural graphs, partition each graph.

4.3.4 Example

We show an application of our method to a program fragment as an example. The following clause is a part of an n-queen program. The predicate mode of **add**, **sub**, **equal**, and **chk** are [++,++,--],[++,++,--], [++,++,--] and [++,++,?,?,?,?,?,?], respectively.

```
check(P, D, L, [Q|Lp0], Lp, A0, A):-
    add(Q, D, Sum @ [any(1)])@ [to(1), sequence(1)],
    equal(Sum, P, R1 @ [to(1)])@ [to(1), sequence(2)],
    sub(Q, D, Dif @ [any(2)])@ [to(2), sequence(1)],
    equal(Dif, P, R2 @ [to(2)])@ [to(2), sequence(2)],
    chk(R1, R2, P, D, L, Lp0, Lp, A0, A) @ [to(2), sequence(3)].
```

Figure 3 shows a diagram of data flow and load partitioning specified by the annotations in the above list. We can see that the graph partitioning is done according to the data flow.

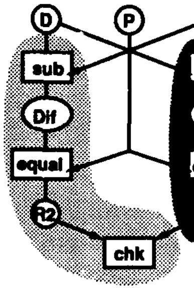
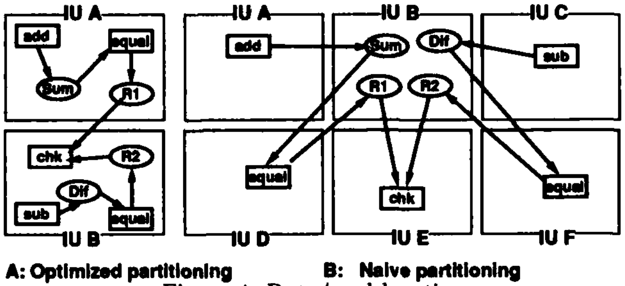


Figure 3: Load partitioning



A: Optimized partitioning B: Naive partitioning

Figure 4: Data/goal location

Figure 4:A shows the resulting allocation. The large square represent each IU, and arcs between goals and data represent data references. Figure 4:B shows the worst allocation caused by naive load partitioning; it separates all the goals and locates all the data on the local IU. By optimal partitioning, only one remote data reference takes place. In contrast, all the memory accesses are remote with naive partitioning. Note that concurrency is not reduced by the optimized partition, even though only two IUs are used.

4.4 Preliminary Evaluation

We have done preliminary evaluation to gain some understanding about the relation between program concurrency and the effect of our dynamic/static distribution method.

We customized a fleng interpreter on PIE64 for evaluation. The interpreter distributes goals and data as specified by the annotations. It shares the run-time kernel with the compiler system, and has the dynamic load-distribution facility mentioned above in Section 3. The facility can be disabled to evaluate the facility itself.

For the evaluation, we used a well-known program "primes", which finds all prime numbers less than 200. This program has relatively low concurrency; 15 - 20 on average. We executed several of "primes" simultaneously, to get variation in concurrency.

Among the "primes", there are no communication, therefore, the absolute value of locality will be different between the result of this experiment and real applications. However, the tendency of the locality against the program concurrency will be same.

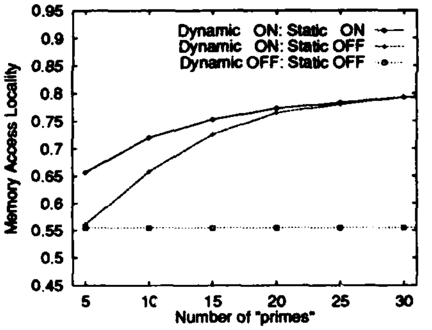


Figure 5: Locality of primes

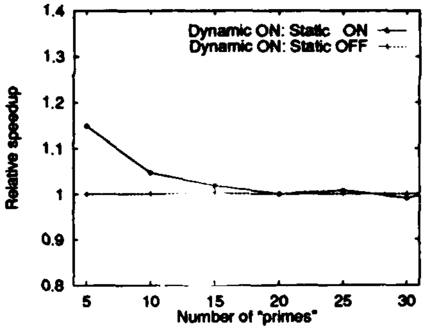


Figure 6: Relative speed up

Figure 5 shows the memory reference locality against the number of "primes". The lowest graph shows the most naive method, i.e. no dynamic load distribution and no static load partitioning. It always shows same low locality. The middle graph shows the result of the dynamic load-distribution. When the concurrency is high, it shows enough locality, however, the method shows no improvement in low concurrency area. The highest graph shows the result of the combination of the dynamic load distribution and the static load partioning. The graph keeps highest locality among three.

The higher the concurrency becomes, the smaller the difference between the two graphs. When all other IUs are loaded, the run-time kernel does not distribute any loads to suppress excessive concurrency. Therefore, when the concurrency is high, the effect of the dynamic load distribution dominates, and it hides the effect of static partitioning.

Figure 6 shows the relative speed up of statically load partitioned programs compared with the naive one, against the number of "primes". The dynamic load-distribution facility is enabled in both case. Note that the reduction speed of interpreter is low, so the remote access penalty appears to be relatively low. As implied by the memory reference locality, the higher the concurrency becomes, the smaller the relative speed up.

From the results described above, we conclude that; (1) the dynamic load distribution is not enough in low-concurrency area, (2) the static load-partitioning is effective to fill up the weak point of the dynamic load distribution.

5 Conclusion and future work

We have described a fleng compiler and its static load partitioning and scheduling based on data-flow analysis. Static load partitioning is effective, especially in the low concurrency area.

For future work, the following are considered to be important:

- Using profiling data in static scheduling,
- Adopting more complicated static analysis,
- Evaluating our whole system.

Here, we only use static data. However, for selecting targets from the several longest walks, simple profiling data will be useful.

Our mode analysis is relatively naive and simple, and it restricts static optimization. We are planning to establish more complicated mode analysis. With that analysis, other static optimization is possible; for example, grain combination.

Finally, evaluation of our whole system is required, including the run-time kernel.

Acknowledgments

This work has been supported by Grant-in-Aid for Scientific Research (No.62065002, No.03555071, No.03003891) from the Ministry of Education, Science and Culture.

We are thankful to Professor Randy Goebel for editing an earlier draft of this paper.

References

[1] Takashi CHIKAYAMA. Operating system PIMOS and kernel language KL1. In *Proc. of International Conference on FIFTH GENERATION COMPUTER SYSTEMS 1992*, pages 73–88, 1992.

[2] Yasuo HIDAKA, Hanpei KOIKE, Jun'ichi TATEMURA, and Hidehiko TANAKA. A static load partitioning method based on execution profile for committed choice languages. In *Proc. of ILPS*, 1991.

[3] Ehud SHAPIRO, editor. *Concurrent Prolog*. The MIT Press, 1987.

[4] Kentaro SHIMADA, Hanpei KOIKE, and Hidehiko TANAKA. UNIRED II: The high performance inference processor for the parallel inference machine PIE64. In *Proc. of Int. Conf. of Fifth Generation Computer Systems*, pages 715–722, 1992.

PART IX

FUNCTIONAL LANGUAGES, DATAFLOW MODELS AND IMPLEMENTATION

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Compiling Higher-Order Functions for Tagged-Dataflow

P. Rondogiannis[a] amd W. W. Wadge[b]

[a]Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, B.C., CANADA V8W 3P6, prondo@csr.uvic.ca

[b]Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, B.C., CANADA V8W 3P6, wwadge@csr.uvic.ca

Abstract: The implementation of higher-order functions on tagged-dataflow machines, has always been a problematic issue. This paper presents and formalizes an algorithm for transforming a significant class of higher-order programs, into a form that can be executed on a dataflow machine. The meaning of the resulting code is described in terms of Intensional Logic, a mathematical formalism which allows expressions whose value depends on hidden contexts.

Keyword Codes: D.1.1; D.3.1; F.4.1
Keywords: Applicative (Functional) Programming; Programming Languages, Formal Definitions and Theory; Mathematical Logic

1 Introduction

One of the most appealing features of the dataflow model of computation, is its close relationship with functional programming. This is evidenced by the fact that all the well-known dataflow languages are functional in nature [1]. Moreover, it is generally easy to implement first-order functions on a tagged-token dataflow machine. This is achieved through the use of appropriate tag manipulation operations, which can be thought of as having a "colouring" effect on tokens [2]. However, this scheme fails when higher-order functions are considered. In practice, higher-order functions are implemented using non-dataflow mechanisms such as *closures* [3], an approach which is against the basic principles and spirit of tagged-dataflow.

This paper considers the implementation of a significant subset of a higher-order functional language, using only simple dataflow concepts (such as *tags*). Given a program of order N, the technique gradually transforms it into a zero-order program extended with appropriate context (tag) manipulation operators. The algorithm is initially presented at an informal level and illustrated by examples. The last sections of the paper contain a formal definition of the transformation algorithm. The paper concludes by briefly discussing implementation issues and possible extensions.

2 The First-Order Case

Before considering higher-order programs, we outline the approach we adopt for the first-order case; this was initially developed in [16] and also described in [6]. The algorithm given in [16] transforms a first-order program into a set of zero-order definitions that contain context manipulation operations. As the semantics of the resulting code is based on Montague's Intensional Logic [10], the resulting definitions are also referred in [16] as *intensional* definitions. The functional language adopted in [16] is ISWIM [9]. Programs are initially flattened using a technique similar to lambda-lifting [7]. The following algorithm is then applied to this flattened code.

1. Let f be a function appearing in the program. Number the textual occurrences of calls to f starting at 1.
2. Replace the ith call of f by $call_i(f)$.
3. Remove the formal parameters from the definition of f, so that f is defined as an ordinary individual variable.
4. Introduce a definition for each formal parameter of f. The right hand side of the definition is the operator *actuals*, applied to a list of the actual parameters corresponding to the formal parameter in question. The actual parameters are listed in the order in which the calls are numbered.

As an example, consider the following program:

$$\begin{aligned} result &= f(5) \\ f(x) &= inc(x+1) \\ inc(y) &= y+1 \end{aligned}$$

The following zero-order intensional program is obtained, when the algorithm is applied:

$$\begin{aligned} result &= call_1(f) \\ f &= call_1(inc) \\ inc &= y+1 \\ x &= actuals(5) \\ y &= actuals(x+1) \end{aligned}$$

An execution model is established by considering the $call_i$ and *actuals* as operations on finite lists of natural numbers (referred from now on as *tags* or *contexts*). Execution of the program starts by demanding the value of the variable *result* of the intensional program, under the empty tag [ ]. The operator $call_i$ augments a tag t by prefixing it with i. On the other hand, *actuals* takes the head i of a tag, and uses it to select its ith argument. Formally, the semantic equations as introduced in [16], are: [1]

$$\begin{aligned} (call_i(A))_t &= A_{[i|t]} \\ (actuals(A_1,\ldots,A_n))_{[i|t]} &= (A_i)_t \\ c(A_1,\ldots,A_n)_t &= c((A_1)_t,\ldots,(A_n)_t) \end{aligned}$$

[1] The notation $[i|t]$ denotes a list with head i and tail t.

where $A, A_1, \ldots, A_n$ are values in the target (intensional) language and c is an n-ary operation symbol. In particular, individual constants in the target language have the same value under any tag, e.g. $(2)_t = 2$, for all t. The evaluation of the program proceeds by applying the above semantic rules; every time a variable is encountered, it is replaced by its defining expression (for an example execution, see [12]). The technique just described has been extensively used in the implementations of the Lucid functional-dataflow language [15] as well as in other Lucid-related languages and systems.

3 The Higher-Order Case

The main idea for the generalization of the technique to higher-order programs, was initially proposed in [14] and has since been extended and formalized in [11]. In this section we outline how the technique can be applied to a significant class of higher-order programs. Intuitively, the language we adopt allows a Pascal-like use of higher-order objects:

1. Function names can be passed as parameters but not returned as results.

2. Operation symbols are first-order.

The main idea of the generalized transformation is that an N-order functional program can first be transformed into an $(N-1)$-order intensional program, using a similar technique as the one for the first-order case. The same procedure can then be repeated for the new program, until we finally get a zero-order intensional program.

The idea of tags is now more general: for a program of order N, a tag is an N-tuple of lists, where each list corresponds to a different order of the program. The operators are also more general as they have to manipulate the new, more complicated contexts. As the transformation for the higher-order case consists of a number of stages, we use a different set of operators for each stage. For the first step we use the operators $actuals_N$ and $call_{(N,i)}$, where i ranges as in the first-order case. For the second step, we use $actuals_{(N-1)}$ and $call_{(N-1,i)}$, and so on.

The code that results from the transformation can be executed following the same basic principles as in the first-order case. In the following, we present the transformation algorithm and describe the semantics of the generalized operators. Consider the following simple second-order program:

$$\begin{aligned} result &= apply(inc, 10) \\ apply(f, x) &= f(x) \\ inc(y) &= y + 1 \end{aligned}$$

The function *apply* is second-order because its first argument is first-order. The generalized transformation, in its first stage eliminates the first argument of *apply*:

$$\begin{aligned} result &= (call_{(2,1)}(apply))(10) \\ apply(x) &= f(x) \\ inc(y) &= y + 1 \\ f &= actuals_2(inc) \end{aligned}$$

We see that the program that resulted above is first-order: all the functions have zero-order arguments. The only exception is the definition of f which is an equation between function expressions. We can easily change this by introducing an auxiliary variable z:

$$\begin{aligned} result &= (call_{\langle 2,1\rangle}(apply))(10) \\ apply(x) &= f(x) \\ inc(y) &= y+1 \\ f(z) &= (actuals_2(inc))(z) \end{aligned}$$

It is necessary to pass z inside the *actuals* before performing the next stage of the transformation. However, the *actuals* operator alters (*shortens*) the tags. In order for z to be evaluated in the outer tag, it has to be appropriately advanced before entering the scope of *actuals*. This is done as follows:

$$\begin{aligned} result &= (call_{\langle 2,1\rangle}(apply))(10) \\ apply(x) &= f(x) \\ inc(y) &= y+1 \\ f(z) &= actuals_2(inc(call_{\langle 2,1\rangle}(z))) \end{aligned}$$

This completes the first stage of the transformation. Now, we have a first-order intensional program, and we can apply the technique for the first-order case, which gives the final program:

$$\begin{aligned} result &= call_{\langle 1,1\rangle}(call_{\langle 2,1\rangle}(apply)) \\ apply &= call_{\langle 1,1\rangle}(f) \\ inc &= y+1 \\ f &= actuals_2(call_{\langle 1,1\rangle}(inc)) \\ z &= actuals_1(x) \\ y &= actuals_1(call_{\langle 2,1\rangle}(z)) \\ x &= actuals_1(10) \end{aligned}$$

The (informal) algorithm for the higher-order case consists of repeating the following steps until the program becomes zero-order.

1. Let f be a function of the current highest order d. Number the textual occurrences of calls to f starting at 1.

2. Remove from the ith call to f all the actual parameters of order $(d-1)$. Prefix the call to f with $call_{\langle d,i\rangle}$.

3. Remove from the definition of f the formal parameters of order $(d-1)$.

4. For every formal parameter x of f that was eliminated, introduce an $actuals_d$ definition, using the same procedure as in the first-order case.

5. Introduce new variables according to the type of x. Add the variables to both sides of the definition of x, appropriately advancing them before they enter the scope of $actuals_d$.

In the execution model for a program of order N, tags are N-tuples of lists of natural numbers, and each list corresponds to a different order of the initial program (or equivalently, a different stage in the transformation). We will use the notation $< t_1, ..., t_N >$ to denote a tag. The operators *call* and *actuals* can now be thought of as operations on these more complicated tags. The semantics of $call_{\langle d,i\rangle}$ can be described as follows: given a tag, d is used in order to select the corresponding list from the tag. The list is then prefixed with i and returned to the tag.

On the other hand, $actuals_d$ takes from the tag the list corresponding to d, uses its head i to select the ith argument of *actuals*, and returns the tail of the list to the tag. The new semantic equations are:

$$\begin{array}{rcl} (call_{(d,i)}(A))_{<t_1,\ldots,t_d,\ldots,t_N>} & = & A_{<t_1,\ldots,[i|t_d],\ldots,t_N>} \\ (actuals_d(A_1,\ldots,A_n))_{<t_1,\ldots,t_d,\ldots,t_{,}.>} & = & (A_{head(:_d)})_{<t_1,\ldots,tail(t_d),\ldots,t_N>} \end{array}$$

For n-ary operation symbols c, the semantic equation is the same as in the first-order case. The evaluation of a program starts with an N-tuple of empty lists, one for each order. Execution proceeds as in the first-order case, the only difference being that the appropriate list within the tuple is accessed every time.

4 Extensional and Intensional Languages

In this section, the syntax of the source and target languages are introduced. Based on this material, a formal definition of the transformation algorithm is presented in the next section.

4.1 The Source Functional Language

The source language adopted is a simple functional language, of recursive equations, whose syntax satisfies the two requirements introduced in the previous section. The source language will also be referred as the *extensional* language, to distinguish it form the target *intensional* one. We proceed by defining the set of allowable types of the language and then presenting its formal syntax.

Definition 4.1 The set **Typ** of extensional types is ranged over by τ and is recursively defined as follows:

- The base type ι is a member of **Typ**.
- If $\tau_1,\ldots,\tau_n$ are members of **Typ**, then $(\tau_1 \times \cdots \times \tau_n) \to \iota$ is a member of **Typ**.

For simplicity, when $n = 0$, we assume that $(\tau_1 \times \cdots \times \tau_n) \to \iota$ is the same as ι. The meaning of the base type ι is a given domain $\mathbf{V}$, which will be called the *extensional domain.*

Definition 4.2 The syntax of the extensional functional language *Fun* is described by the following rules:

- $\mathbf{Var}^\tau$ are given sets of *variable* symbols of type τ and are ranged over by f^τ.
- $\mathbf{Con}^\tau$ are given sets of *operation* symbols of type τ, ranged over by c^τ.
- **Exp** is a set of *expressions* ranged over by E and given by:

$$E \ ::= \ f^\tau \ | \ (c^{(\iota\times\cdots\times\iota)\to\iota}(E_1^\iota,\ldots,E_n^\iota))^\iota \ | \ (f^{(\tau_1\times\cdots\times\tau_n)\to\iota}(E_1^{\tau_1},\ldots,E_n^{\tau_n}))^\iota$$

The set of expressions of type τ is denoted by $\mathbf{Exp}^\tau$.

- **Def** is a set of *definitions* ranged over by D and given by:

$$D \ ::= \ (f^{(\tau_1\times\cdots\times\tau_n)\to\iota}(x_1^{\tau_1},\ldots,x_n^{\tau_n}) \doteq E^{\iota})$$

- **Prog** is a set of *programs* ranged over by P. A program is a set of definitions, exactly one of which defines the nullary variable *result*. The variables appearing in the program (including formal parameters), are distinct from each other. Function definitions do not contain global nullary variable symbols in their body.

The order of a function is the order of its type. Formally:

Definition 4.3 The order of a type is recursively defined as follows:

$$\begin{aligned} order(\iota) &= 0 \\ order((\tau_1 \times \cdots \times \tau_n) \to \iota) &= 1 + max(\{order(\tau_k) \mid k = 1,\ldots,n\}) \end{aligned}$$

4.2 The Target Intensional Language

Intensional Logic [10, 5] is a mathematical formal system which allows expressions whose value depends on hidden contexts. From the informal exposition given in sections 2 and 3, it is clear that the programs that result from the transformation, can be evaluated with respect to a context. The zero-order variables that appear in the final program, are not ordinary variables that have a fixed data value. In fact, they can intuitively be thought of as tree-like structures, that have individual data values at their nodes. Such variables are also called *intensions* [5].

Another way one can think about intensions, is to consider them as functions from contexts to data values, i.e. as members of ($\mathbf{W} \to \mathbf{V}$), where **W** is a set of contexts (also called *possible worlds*) and **V** is a data domain. Given an intension and a context, the value of the intension at this context can be computed. Moreover, operations like $+$, $*$, $if - then - else$ and so on, are now operations that take as arguments intensions. For example, if x and y are intensions, then $(x + y)$ can be thought of as a new intension, whose value at every context is the sum of the values of the intensions x and y at this same context. In other words, operations on intensions are defined in a pointwise way using the corresponding (extensional) operations on the data domain **V**. The above discussion is formalized by the following definitions:

Definition 4.4 The set **ITyp** of intensional types is ranged over by τ and is recursively defined as follows:

- The type ι^* is a member of **ITyp**.
- If $\tau_1,\ldots,\tau_n$ are members of **ITyp** then $(\tau_1 \times \cdots \times \tau_n) \to \iota^*$ is a member of **ITyp**.

Let **W** be a set of *possible worlds* (or *contexts*, or *tags*) and let **V** be an extensional domain. Then, the meaning of the base type ι^* is the domain $\mathbf{I} = (\mathbf{W} \to \mathbf{V})$, called the ***intensional domain***. Members of **I** are called ***intensions***. The set **W** plays the most crucial role in the specification of an intensional language. In our case, $\mathbf{W} = (\mathbf{N} \to \mathbf{List(N)})$, i.e., **W** is the set of functions from natural numbers to lists of natural numbers. Given an extensional language Fun over the data domain **V**, an intensional language $In(Fun)$ can be created as follows:

Definition 4.5 The syntax of the intensional language $In(Fun)$ is described by the following rules:

- **IVar**$^\tau$ are given sets of *intensional variable* symbols ranged over by f^τ.

- **ICon**$^\tau$ is a given set of continuous *intensional operation* symbols, ranged over by c^τ. Members of **ICon**$^\tau$ are also called *pointwise* operations because their meaning is defined in a pointwise way in terms of their extensional counterparts.

- **INp** is a set of non-pointwise operation symbols given by:

$$\{call_{(d,i)} \mid d, i \in \mathbf{N}\} \cup \{actuals_d \mid d \in \mathbf{N}\}$$

- **IExp** is a set of *intensional expressions* ranged over by E and given by:

$$\begin{array}{rcl} E & ::= & f^\tau \mid (c^{(\iota^* \times \cdots \times \iota^*) \to \iota^*}(E_1^{\iota^*}, \ldots, E_n^{\iota^*}))^{\iota^*} \mid (E_0^{(\tau_1 \times \cdots \times \tau_n) \to \iota^*}(E_1^{\tau_1}, \ldots, E_n^{\tau_n}))^{\iota^*} \\ & \mid & (call_{(d,i)}(E^\tau))^\tau, \mid actuals_d(\{i_1 : E_{i_1}^\tau, \ldots, i_n : E_{i_n}^\tau\})^\tau,\ d, i, i_1, \ldots, i_n \in \mathbf{N} \end{array}$$

- **IDef** is a set of *intensional definitions* ranged over by D and given by:

$$D \ ::= \ (f^{(\tau_1 \times \cdots \times \tau_n) \to \iota^*}(x_1^{\tau_1}, \ldots, x_n^{\tau_n}) \doteq E^{\iota^*})$$

- **IProg** is a set of *intensional programs* ranged over by P. The same assumptions are adopted as in the case of extensional programs.

Notice that the *actuals* operator is more general than the one described in section 3. The new semantic equation is:

$$actuals_d(\{i_1 : A_{i_1}, \ldots \ , i_n : A_{i_n}\})(\langle t_1, t_2, \ldots \rangle) = A_{hd(t_d)}(\langle t_1, t_2, \ldots, tl(t_d), \ldots \rangle)$$

The order of intensional functions can be defined in the same way as in the extensional case.

5 A Formal Definition of the Transformation

In this section, we describe the transformation of a d-order program into a $(d-1)$-order one. The following conventions are adopted:

Definition 5.1 Let P be a program and let f be a function symbol defined in P. Then:

- The notation $\overline{f}$ is used to denote an expression of the form $(call_{(d_1,i_1)} \cdots call_{(d_k,i_k)}(f))$, $k \geq 0$. For $k = 0$, $\overline{f}$ denotes the identifier f itself.

- Given $d \in \mathbf{N}$, $pos(f, d)$ is the list of positions of the formal parameters of f that have order less than $(d-1)$. For example, $pos(apply, 2) = [2]$, because the second argument of the *apply* function of section 3, has ordei less than 1.

- The set of $(d-1)$-order formal parameters of f, is represented by $high(f, d)$.

- The set of subexpressions appearing in definitions of the program P is denoted by $Sub(P)$, and the set of function identifiers defined in P is denoted by $func(P)$.

Moreover, we assume the existence of a function $\lceil \cdot \rceil$ that assigns unique natural number identities to expressions, i.e., if $E_1 \neq E_2$ then $\lceil E_1 \rceil \neq \lceil E_2 \rceil$. The elimination of the $(d-1)$-order arguments from function calls, is accomplished with the $\mathcal{M}_d$ function, defined below:

$$\frac{E = \overline{f}}{\mathcal{M}_d(E) = \overline{f}}$$

$$\frac{E = c(E_1, \ldots, E_n)}{\mathcal{M}_d(E) = c(\mathcal{M}_d(E_1), \ldots, \mathcal{M}_d(E_n))}$$

$$\frac{E = (\overline{f^\tau})(E_1, \ldots, E_n),\ order(\tau) = d,\ pos(f,d) = [i_1, \ldots, i_k]}{\mathcal{M}_d(E) = (call_{(d, \lceil E \rceil)}(\overline{f}))(\mathcal{M}_d(E_{i_1}), \ldots, \mathcal{M}_d(E_{i_k}))}$$

$$\frac{E = (\overline{f^\tau})(E_1, \ldots, E_n),\ order(\tau) \neq d}{\mathcal{M}_d(E) = (\overline{f})(\mathcal{M}_d(E_1), \ldots, \mathcal{M}_d(E_n))}$$

$$\frac{E = actuals_{d'}(\{i_1 : E_{i_1}, \ldots, i_n : E_{i_k}\})}{\mathcal{M}_d(E) = actuals_{d'}(\{i_1 : \mathcal{M}_d(E_{i_1}), \ldots, i_n : \mathcal{M}_d(E_{i_n})\})}$$

The first and second rules above are self-explanatory. The third rule applies in the case where a function call is encountered, and the corresponding function is d-order. In this case, the arguments that cause the function to be d-order (i.e., the $(d-1)$-order ones), are removed, and the call is prefixed by the appropriate intensional operator. Notice, that if two function calls in the program are the same, then their translations according to $\mathcal{M}_d$ are identical. The fourth rule applies in the case where the function under consideration is not d-order. In this case, the translation proceeds with the actual parameters of the function call. The final rule concerns the *actuals* operator, and is also straightforward.

The elimination of the $(d-1)$-order formal parameters from the function definitions, is accomplished by the function $\mathcal{R}_d$ defined below:

$$\frac{D = (f(x_1, \ldots, x_n) \doteq E),\ pos(f,d) = [i_1, \ldots, i_k]}{\mathcal{R}_d(D) = (f(x_{i_1}, \ldots, x_{i_k}) \doteq \mathcal{M}_d(E))}$$

For every formal parameter that has been eliminated, a new definition is added to the program. This is achieved using the function $\mathcal{A}_d^f$:

$$\mathcal{A}_d^f(P) = \bigcup_{x_k \in high(f,d)} \{x_k \doteq actuals_d(\{(\lceil F \rceil : \mathcal{M}_d(E_k)) \mid F \equiv (\overline{f})(E_1, \ldots, E_n) \in Sub(P)\})\}$$

The overall translation of a d-order program into a $(d-1)$-order one, is performed by the function $\mathcal{T}_d$:

$$\mathcal{T}_d(P) = (\bigcup_{f \in func(P)} \mathcal{A}_d^f(P)) \cup (\bigcup_{D \in P} \{\mathcal{R}_d(D)\})$$

The translation described above, introduces certain *actuals* definitions, whose result type is not first order. Let

$$f^\tau \doteq actuals_d(\{i_1 : E_{i_1}, \ldots, i_n : E_{i_n}\})^\tau$$

be one of them, and suppose that $\tau = \tau_1 \times \cdots \times \tau_n \rightarrow \iota^*$. We introduce n new variables $z_1^{\tau_1}, \ldots, z_n^{\tau_n}$ and equivalently rewrite the above definition as:

$$f(z_1, \ldots, z_n) \doteq actuals_d(\{i_1 : E_{i_1}, \ldots, i_n : E_{i_n}\})(z_1, \ldots, z_n)$$

It can be shown [11] that the above definition is equivalent to the following one in which the parameters have entered the scope of *actuals*:

$$\begin{aligned} f(z_1, \ldots, z_n) \doteq actuals_d(\{ \quad & i_1 : E_{i_1}(call_{\langle d,i_1\rangle}(z_1), \ldots, call_{\langle d,i_1\rangle}(z_n)), \ldots, \\ & i_n : E_{i_n}(call_{\langle d,i_n\rangle}(z_1) \ldots, call_{\langle d,i_n\rangle}(z_n)) \ \}) \end{aligned}$$

In the program that results following this approach, all the definitions have first-order result types. The transformation algorithm can therefore be applied repeatedly, until a zero-order program is obtained.

6 Discussion and Conclusions

An algorithm for transforming higher-order functional programs into zero-order intensional ones has been presented. The resulting code can be evaluated relative to a context, giving in this way a method of implementing a class of higher-order functions in a purely dataflow manner.

Two preliminary implementations have been undertaken by the authors. In [12], a hashing-based approach is adopted. More specifically, identifiers together with their context and value, are saved in a Value-Store. If the identifier is encountered again under the same context during execution, the Value-Store is looked-up using a hashing function, and the corresponding value is returned. The measurements reported in [12] indicate that the technique can compete with modern graph-reduction [8] implementation techniques for functional languages. However, hashing is reported as a factor that can cause a severe bottleneck to the implementation.

In [13], a technique is proposed in which contexts are appropriately incorporated in the headers of activation records. A similar approach is reported in [4]. The main advantage of an activation-record based technique, is that hashing is avoided, and a significant gain in execution time is obtained. The results presented in [13] indicate that the tag-based code outperforms many well-known reduction-based systems.

Future work includes investigating optimizations such as *strictness analysis* as well as intensional code transformations. Also, we are currently considering the extension of the technique for functions with higher-order result types.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council of Canada and a University of Victoria Graduate Fellowship.

References

[1] W. B. Ackerman. Data Flow Languages. *IEEE Computer*, pages 15–24, Feb. 1982.

[2] Arvind and D. Culler. Dataflow architectures. In S. S. Thakkar, editor, *Selected Reprints on Dataflow and Reduction Architectures*, pages 79–101. IEEE Computer Society Press, 1987.

[3] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, Mar. 1990.

[4] D. Culler and G. Papadopoulos. The Explicit Token Store. *Journal of Parallel and Distributed Computing*, 10:289–308, 1990.

[5] D. Dowty, R. Wall, and S. Peters. *Introduction to Montague Semantics.* Reidel Publishing Company, 1981.

[6] A. A. Faustini E. A. Ashcroft and R. Jagannathan. An Intensional Language for Parallel Applications Programming. In B.K.Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 11–49. ACM Press, 1991.

[7] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proceedings of IFIP Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, 1985.

[8] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[9] P. J. Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3):157–166, Mar. 1966.

[10] R. Montague. *Formal Philosophy, Selected Papers of R. Montague.* Yale University Press, 1974.

[11] P. Rondogiannis. *Higher-Order Tagged Dataflow.* PhD thesis, Department of Computer Science, University of Victoria, Canada, 1994. in preparation.

[12] P. Rondogiannis and W. Wadge. A Dataflow Implementation Technique for Lazy Typed Functional Languages. In *Proceedings of the Sixth International Symposium on Lucid and Intensional Programming*, pages 23–42, 1993.

[13] P. Rondogiannis and W. Wadge. Higher-Order Dataflow and its Implementation on Stock Hardware. In *Proceedings of the ACM Symposium on Applied Computing*, pages 431–435. ACM Press, 1994.

[14] W. W. Wadge. Higher-Order Lucid. In *Proceedings of the Fourth International Symposium on Lucid and Intensional Programming*, 1991.

[15] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language.* Academic Press, 1985.

[16] A. A. Yaghi. *The Intensional Implementation Technique for Functional Languages.* PhD thesis, Department of Computer Science, University of Warwick, Coventry, UK, 1984.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Dataflow-Based Lenient Implementation of a Functional Language, *Valid*, on Conventional Multi-Processors

Shigeru Kusakabe, Eiichi Takahashi, Rin-ichiro Taniguchi and Makoto Amamiya

Department of Information Systems, Kyushu University, Kasuga Fukuoka 816, Japan

Abstract: In this paper, we present a dataflow-based lenient implementation of a functional language, *Valid*, on conventional multi-processors. A data-flow execution scheme offers a good basis to execute in a highly concurrent way a large number of fine grain function instances, created during the execution of a functional program. The lenient execution and split-phase operation will overlap the idle time caused by remote memory access and remote calls. However, it is necessary to reduce the overhead to handle fine-grain parallelism on conventional multi-processors with no special hardware for fine-grain data/message-flow processing. We discuss compilation issues of dataflow-based implementation and runtime systems to support fine-grain parallel execution on two different types of conventional multi-processor: a shared-memory multi-processor, Sequent Symmetry S2000, and a distributed-memory multi-processor, Fujitsu AP1000. We also show the preliminary evaluation of our implementation.

Keyword Codes: D.1.1; D.1.3; D.3.4
Keywords: Functional Programming; Concurrent Programming; Programming Language Processor

1 Introduction

Despite the development of massively parallel computer architectures in recent years, writing programs for such machines is still difficult, because most programming languages still assume a sequential computation scheme. Explicit descriptions of procedural-based parallel execution, and mapping to a particular architecture require a high degree of skill. With implicit parallel programming languages, which promote the productivity of software for parallel computer systems, compilers and runtime systems can easily exploit parallelism without explicit descriptions of parallel execution.

Functional programming languages have various attractive features, due to their pure functional semantics, for writing short and clear programs, providing the ability to abstract architectural peculiarity and promoting programming productivity. It is easy for both the programmer and the implementation to reason about functional programs both formally and informally. The merits are more explicit in writing programs for massively parallel processing, since a functional program is constructed as an ensemble of functions and their instances can be executed as concurrent processes without interfering with each other.

In this paper, we present a dataflow-based lenient implementation of a functional language, *Valid*, on conventional multi-processors. A data-flow based execution scheme offers a good basis to execute in a highly concurrent way a number of fine grain function instances, dynamically created during the execution of a functional program. Lenient execution and split-phase operation will overlap the idle time caused by remote memory access and remote calls. However, it is necessary to reduce the overhead to handle fine-grain parallelism on conventional multi-processors with no special hardware for fine-grain data/message-flow processing.

In section2, we discuss compilation issues for our dataflow-based implementation, and present a dataflow-based intermediate code and an overview of our multi-thread[1] execution. Our implementation targets are two different types of conventional multi-processor. In section 3, implementation issues and a preliminary evaluation for a shared-memory multi-processor, Sequent Symmetry S2000, and in section 4, those for a distributed-memory multi-processor, Fujitsu AP1000, are presented.

2 Compilation

The compiler consists of a machine independent phase and a machine dependent phase. The machine independent phase generates a dataflow-based intermediate code. Then the machine dependent phase optimizes the intermediate code for each target machine.

2.1 Intermediate Code

The intermediate code consists of Datarol[2] graph code and SST(Source level Structure Tree). Datarol reflects dataflows inherent in a source program. It is designed to remove redundant dataflows and make code scheduling easy by introducing a concept of by-reference data access. The synchronized memory access employs an I-structure mechanism[3]. Datarol instructions are similar to conventional three-address instructions, except that each instruction specifies its succeeding instructions explicitly. SST represents the source program structure and helps to determine the optimal grain size and code scheduling which can utilize the locality.

Table 1 shows the extracts from Datarol instruction set. Details of SST and synchronized memory access operations are omitted because of space limitation. A general Datarol instruction is expressed as

$$l\!:\![*]\ \texttt{op}\ r_{S1}\ r_{S2}\ r_D\ \texttt{->}\ D$$

where l is a label of this instruction and D, or continuation point, is a set of labels of instructions which should be executed successively. This instruction means that op operates on two operands r_{S1} and r_{S2}, and stores the result in r_D , which generally denotes a register name. The optional tag '*' ([] means that what is enclosed is optional.) indicates that a partner check is

Table 1: Extracts from Datarol instructions

instruction	semantics
op r_{S1} r_{S2} r_D -> D	conventional 3-address arithmetic and logical instructions
sw r_{S1} -> D_t, D_f	branch
call r_{S1} r_D -> D	activate new instance
link r_{S1} r_{S2} *slot*	link r_{S2} value to r_{S1}
rlink r_{S1} r_D *slot*->D	link r_D address to r_{S1}
receive r_{S1} *slot*-> D	receive parameter, r_{S1}
return r_{S1} r_{S2}	return r_{S2} value to r_{S1}
rins	release current instance

[1] A thread is an instruction sequence to be executed exclusively.

required for the instruction. The branch instruction SW transfers a control to continuation points D_t or D_f according to its boolean operand. Function application instructions are executed as split-phase transactions. A `call` instruction activates an instance of a function specified by r_{S1}, and sets its instance name to r_D . A `link` instruction sends the value of r_{S2} as the *slot*-th parameter to the instance specified by r_{S1}. An `rlink` instruction sends the current instance name and r_D address as a continuation point. A `receive` instruction receives the *slot*-th value, stores it in r_{S1}, and triggers D. A `return` instruction returns the value of r_{S2} to the continuation point r_{S1} specified by the corresponding `rlink` in the parent instance. An `rins` instruction releases the instance.

2.2 Computation Model for Conventional Multi-Processors

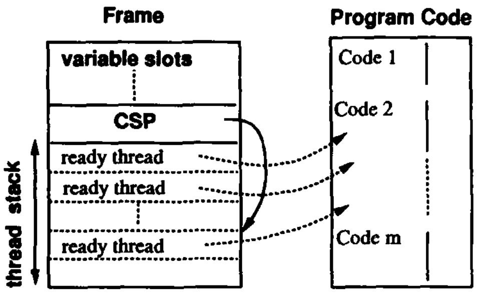


Figure 1: Overview of a frame

Instance level parallelism is exploited by using a data structure, called frame, which is allocated to each function instance. Intra-instance parallelism is exploited as multi-thread parallel processing. Fig.1 shows the overview of a frame. The *variable slots* area is used to manipulate local data. The *thread stack* is provided in order to dynamically schedule intra-instance fine-grain threads. When a thread becomes ready to run, or a *ready thread*, the entry address of the thread is pushed into the thread stack. A processor executes instructions of a thread in a serial order until it reaches its termination point. If the execution of the current thread terminates, a ready thread is popped from the top of the thread stack for the next execution. A processor get a ready thread from a thread stack and execute the thread repeatedly. If ready threads are exhausted, the processor switches to another frame.

In principle, the compiler partitions a Datarol graph at split-phase operation points, traces subgraphs along arcs in a depth-first way and rearranges nodes in the traced order, in order to generate a threaded code for conventional multi-processors[10]. However, the choices of split-phase operation and code scheduling strategy depend on the run-time costs of parallel processing of the target machine.

3 Implementation on a Shared Memory Machine

3.1 Implementation

Fig.2 shows the outline of the multi-task monitor implemented on Sequent Symmetry DYNIX. Frames are realized as data structures in a shared memory. Arguments and local variables are stored in a work area. To reduce the overhead of allocation and deallocation of a frame, each processor has its own free-list of frames. A task pool is a chain of runnable frames. A frame is put into a task pool when it becomes ready to run. Processors get a runnable frame from a task pool in a mutually exclusive way, and execute the frame. The number of task pools is set to be greater than the number of processors in order to reduce the overhead caused by the access race to the task pools. Processors execute intra-instance threads within their local environment, so that the cache mechanism can work effectively, and bus traffic, a main cause of bottlenecks in shared memory multi-processors, can be reduced.

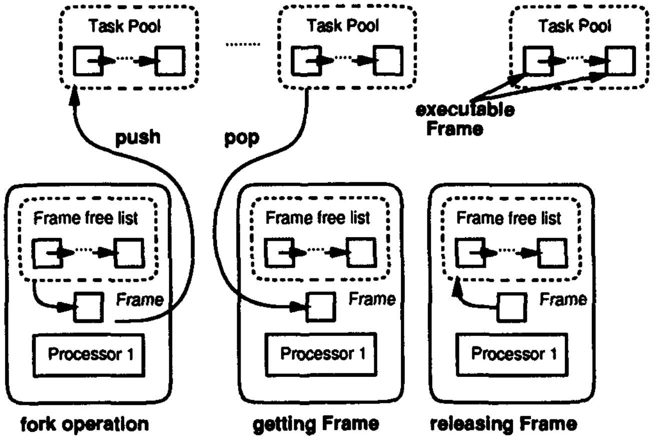


Figure 2: Multi-task monitor for a shared memory machine

An instance frame may have one of three states: *running, ready* or *suspended* state. A running instance is an executing instance. A ready instance is in a stack pool with one or more executable threads in its thread stack. A suspended instance has temporarily no executable thread, but has not completed its computation. At the end of the execution of a thread in a running instance, the synchronizing variables of the continuation threads are decremented. When the variable becomes 0, if the target instance is in suspended state, the instance is pushed into a task pool; if it is in running or ready state, the target thread becomes ready and is pushed into the thread stack of the instance. In Sequent Symmetry, checking synchronizing variables can be done by one operation without context switching. Thus its overhead is not so serious compared with the AP1000, where this checking of synchronizing variables may cause message passing and context switching.

At the end of a thread execution, the processor also tries to execute another thread in the current frame. If there is no other executable thread in the frame, a context switching will occur. When the current instance is completed, the processor puts back the current frame in its own frame free-list to reuse it. In a context switch, the processor abandons the current frame and tries to start another executable frame.

3.2 Performance

We have evaluated the performance on Sequent Symmetry S2000, using 1 to 16 processors. Table 2 shows the elapsed time in seconds for parallel Valid and sequential C programs using the same algorithm, with the relative speedup ratios and the overhead caused by parallel control for Valid programs. The relative speedup ratios are evaluated as: *(time of C program) ÷ (time of Valid program).* The overhead column shows the proportion of system time to total time based on the results from the DYNIX profiler. System

Table 2: Time comparisons of Valid and C; speedups and overheads of Valid.

Program	Time (sec.)		Speedup	Overhead
	Valid 16cpus	C 1cpu		
sum(1, 10^6)	0.0375	0.241	6.44	7.69%
matrix(256)	9.07	38.4	4.23	23.4%
nqueen(10)	4.21	25.3	6.01	15.5%
qsort(10^4)	3.21	12.1	3.77	65.8%

time includes the time spent getting, initializing and releasing frames, and synchronizing operations. Figure 3 shows the relative speedups of Valid programs. In the graph, the horizontal axis shows the number of processors used, the vertical axis the speedup, and the linear proportion line the ideal speedup.

The program 'sum(l, h)' calculates the summation from l to h integers. We implement this program with a parallel expression and a reduction operation. The program is partitioned into relatively large portions and distributed to processors equally, so process creation overhead is less critical.

The program 'matrix(n)' computes the product of two $n \times n$ matrices. Although the speedup is close to linear, the speed is about one-third compared with the ideal one. This is probably caused by the implementation of an array structure. An array in C is a one-dimensional or multidimensional collection of homogeneous values. On the other hand, an array in Valid is one-dimensional and may have a collection of heterogeneous values. Therefore, the implementation of an array in Valid must be more complicated than its implementation in C. It is possible to increase the speed of the Valid programs by introducing the same array specification into Valid.

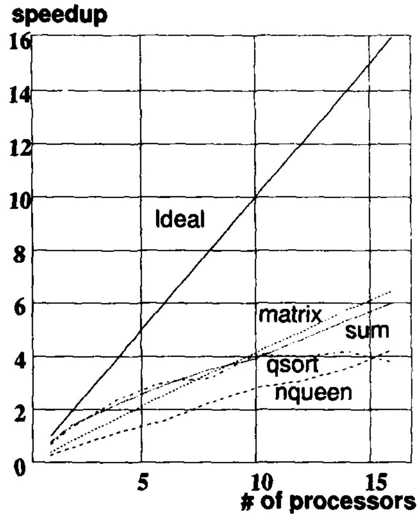


Figure 3: Speedup graphs for Valid programs on Sequent Symmetry S2000

The program 'nqueen(n)' searches all solutions of the n-Queen puzzle. The C program uses library functions to manipulate lists from Valid.

The program 'qsort(n)' rearranges the elements of a list with n integers with a quick sort algorithm. The C program uses library functions from Valid to manipulate lists. In Figure 3, the speedup curve of Valid saturates at about 5.5 after 10 processors.

3.3 Experiment to reduce overhead

To solve the problem of process creation cost in parallel processing, the compiler estimates the cost of each function, and generates a code in which light-weight function applications are not forked, but inline expanded. However, the exact cost of recursive functions cannot be estimated at compile time. While functional programming languages have the advantage that programmers can write programs without paying attention to the parallelism of the program, they have the disadvantage that program optimization is difficult. For example, even if the most effective strategy, that is, determining whether to evaluate in parallel or sequentially, and the most effective mapping of functions and data to processors in multi-computers are obvious to programmers, it is difficult to explicitly express them in programs. As a paradigm to solve the above problem without losing the advantage of functional programming languages, parafunctional programming, such as ParAlfl from Yale university, has been offered [7]. This is a method to extend a functional programming language by introducing meta-linguistic devices such as annotations in the language. We have attempted to extend Valid to optimize the strategy of processing functions, and evaluated its performance.

An extension of function application specification in Valid allows the programmer to express a strategy for deciding between sequential or parallel execution. As a simple example, we present the program 'fib8(n)', an extended version of 'fibo(n)', in which parallel forking is controlled with the argument n.

```
function fib8(n:integer)return(integer)
  = if n<2 then 1
    else fib8(n-1)$[n>=8] + fib8(n-2)$[False];
```

In the above program, if n $\geq$ 8, the function application fib8(n-1) is executed by a parallel fork operation, otherwise sequentially by a local call operation. Table 3 shows the performance improvement on the fib8(30) program.

If the target machine is a multi-computer, the parallel fork control method mentioned above can be expanded easily for mappings of functions and data to processors. By allowing an integer expression to be an expression in $[ ], and by generating a code which regards the value of the expression in $[ ] as a processor ID and maps the function application to a processor, the mechanism of mapped expression in ParAlfl can be implemented.

Table 3: Effect of annotation to control parallel fork operation(16cpus).

Program	Time (sec.)	Overhead
fibo(30)	7.61	73.5%
fib8(30)	2.84	70.6%

4 Implementation on a Distributed Memory Machine

4.1 Implementation

On a distributed-memory machine, AP1000, an instance frame is created by using message passing. The message consists of data, PE ID, frame ID and thread ID. Messages are stored in the system message buffer of the target node. When a PE finds that the current frame has no executable thread, the processor gets a message from its message buffer and switches its context according to the message contents. Message handlers are code fragments generated and inserted in threads by the compiler to handle messages.

Fig.4 illustrates an example of message passing between two frames on different PE nodes. a0,...,a3 and b0,...,b3 represent thread labels. A format of **M.H.xx** in the threads represents a message handler code for a thread **xx**. An arrow from a thread to another thread represents a message passing, and the contents of the message are shown above the arrow.

When PE A executes an operation for instance creation(corresponding to a Datarol call operation) in the thread a0, one PE is selected to activate the new instance(In Fig.4, PE B is selected). An initializing thread, b0, is activated at the beginning of an instance creation. This initializing thread is one of the runtime system threads which require no instance frame. The initializing thread performs **mkFrame** to get a frame area, executes a message handler, and initializes a synchronizing counter to control intra-instance thread scheduling(see **set(b3,2)**). The PE ID and the address of the frame prepared in this execution are packed into a message and sent back to the caller instance. This message tries to activate the threads a1.

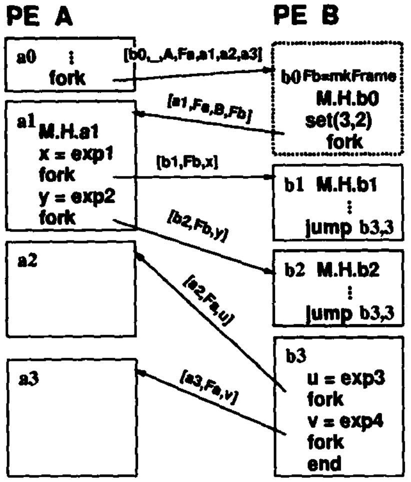


Figure 4: Overview of inter-frame message passing

Two fork operations(corresponding to Datarol links) in the thread a1 will activate the threads b1 and b2. At the heads of the threads b1 and b2, the message handlers extract the data from the message and store the data in the slots of the frame. Concurrently, a1 may proceed its computation and activate some thread successively.

The thread b3 should be activated after the termination of the threads b1 and b2. Synchronization among these threads b1, b2 and b3, is performed by using a synchronizing counter. The value of the synchronizing counter for the thread b3 is realized as a local variable with the initial 2, since the thread b3 should be activated after the termination of the two threads, b1 and b2. In the threads b1 and b2, jump T,S instructions decrement the number of the synchronizing counter and test whether the value of the synchronizing counter is 0 or not. If the value of the synchronizing counter is 0, the thread T is pushed into the thread stack.

After the termination of each thread, PE tries to pop a thread from the thread stack of the current frame to executes it. If the thread stack is empty, PE switches to another frame. The instance activated in the PE B, will terminate after the thread b3. At the end of the execution of the whole instance, **end** releases the allocated instance frame.

4.2 Performance

We report the performance of Valid on the AP1000 using 1 to 64 processors. Table 4 shows the elapsed time in seconds for the Valid and C programs, the relative speedups of Valid programs compared with C programs and the proportion of system library time to the total time. The C programs are written using the same algorithms as the Valid programs.

The library time includes the time spent for message passing operations to send and receive message packets. Figure 5 shows the speedup of Valid programs relative to the number of processors for each benchmark program. In the graph, the horizontal axis shows

Table 4: Time comparisons of Valid and C; speedups and overheads of Valid.

Program	Time (sec.)		Speedup	Task	Library
	Valid	C			
	64cpus	1cpu			
sum(1, 10^6)	0.0385	1.47	38.1	57.1%	42.6%
matrix(512)	10.2	558	54.8	68.7%	31.3%
nqueen(10)	1.57	55.6	35.5	62.9%	37.1%

the number of processors used, the vertical axis the speedup, and the linear proportion line the ideal speedup.

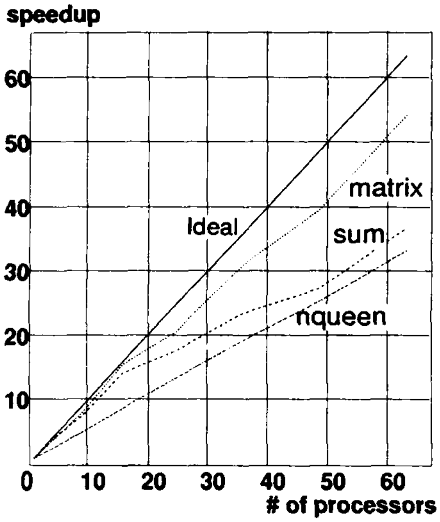
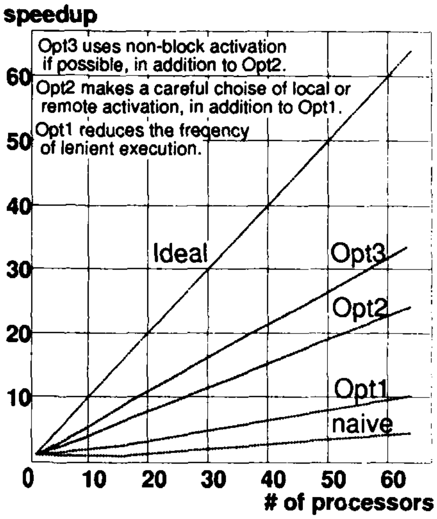


Figure 5: Speedup graphs for Valid programs on AP1000



Opt3 uses non-block activation if possible, in addition to Opt2.

Opt2 makes a careful choise of local or remote activation, in addition to Opt1.

Opt1 reduces the freqency of lenient execution.

Figure 6: Speedup improvement('nq'''en').

As the AP1000 does not have special hardware, a sophisticated compilation technique is necessary for the lenient execution of a fine grain parallel language such as Valid. Our naive implementation showed that communication overheads are critical. We examined the cost of our runtime system, focusing on the cost-hierarchy compilation, especially trying to reduce the frequency of communication and to exploit locality. Fig.6 shows an improvement in the speed-up ratio for the 'nqueen' program. We conclude that a sophisticated exploitation of parallelism is more effective than a naive one.

5 Related Work

Several implementations of functional programming languages on parallel machines have been proposed.

SISAL is also implemented on Sequent Symmetry S2000[6]. We report the comparison results of SISAL and Valid with the same programs in section 3.2 on Sequent Symmetry S2000 using 16 processors. The SISAL programs are compiled with the optimizing SISAL compiler, OSC version 12.9. Table 5 shows the elapsed time in seconds for for SISAL and Valid programs.

Table 5: Time comparisons of Valid and SISAL.

Program	Time (sec.)		(ii) / (i)
	(i)Valid (16cpus)	(ii)SISAL (16cpus)	
sum(1, 10^6)	0.0241	0.0200	0.830
matrix(256)	9.07	3.11	0.343
nqueen(10)	4.21	65.2	15.3
qsort(10^4)	3.21	8.09	2.52

In SISAL, the program 'sum(l, h)' is implemented with a product-form loop and a reduction operation. We implement this program with a parallel expression and a reduction operation. The performance of the Valid program is comparable to the performance of the SISAL program. The performance difference of SISAL and Valid 'matrix(n)' programs is caused by the implementation of an array structure as mentioned in section 3.2. The SISAL 'nqueen(n)' program is implemented with streams and is not parallelized. In the SISAL program, the size of a stream, which contains the solutions, is determined after the search of all solutions is completed, so that it is not invariant. OSC does not concurrentize loop forms which produce variant size streams (and arrays). On the other hand, the Valid program is parallelized. The SISAL 'qsort(n)' is implemented with arrays, and is not parallelized, because of the concurrentization strategy of OSC mentioned above.

The $\langle \nu, G \rangle$-machine[4] is a super-combinator graph-reduction machine. Programs in Lazy ML are translated into a set of definitions of super combinators and the definitions are then compiled in the object code of the $\langle \nu, G \rangle$-machine. In the $\langle \nu, G \rangle$-machine, all the data are in the shared memory as fragments of the program, which form a *program graph* dynamically, since the processing order is determined dynamically at runtime. Frequent access to the shared memory causes bus saturation. Our implementation may have the same problem. However, our implementation on a shared memory can reduce shared memory access by using as many stacks and registers as possible, since access to frames in the shared memory is only to arguments and local variables of a function. Although our current implementation does not support lazy evaluation and higher-order functions, from the viewpoint of efficiency, our implementation has an advantage over the $\langle \nu, G \rangle$-machine.

Our implementation is similar to Threaded Abstract Machine(TAM)[5]. TAM is also based on a thread-level dataflow model, and such dataflow graphs are constructed from Id programs. The execution model of TAM is an extension of a hybrid dataflow model. Our implementation also has its origin in an optimized dataflow, Datarol[2], and our intermediate DVMC can be mapped onto the Datarol-II architecture. However, a sophisticated optimization technique is necessary to implement Valid on conventional multiprocessors.

6 Conclusion

As Valid is a functional programming language originally designed for a dataflow architecture, it is easy to extract parallelism of various level. Implementation issues based on the dataflow scheme for the Sequent Symmetry and the Fujitsu AP1000 were discussed. A function instance is implemented as a concurrent process using a frame. Intra-instance parallelism is realized as multi-threads scheduled in the frame.

Our implementation tries to exploit fine grain parallelism according to the parallel processing capacity of the target machine. However, the grain size of our multi-thread implementation is still too fine for conventional processors. We find that source level annotation to control the grain size, and a sophisticated scheduling method taking into account the parallel processing cost peculiar to the target system, are effective in order to reduce the overhead of fine grain parallel processing.

Acknowledgements

We express our thanks to T.Nagai and Y.Yamashita for discussions and help on our work. We are grateful to Fujitsu Laboratory for providing us with the AP1000.

References

[1] M. Amamiya, et al. : "*Valid*: A High-Level Functional Programming Language for Data Flow Machine", Rev. ECL, Vol.32, No.5, pp.793-802, NTT, 1984.

[2] M.Amamiya, and R.Taniguchi : "Datarol: A Massively Parallel Architecture for Functional Language", Proc. IEEE 2nd SPDP, pp.726-735, 1990.

[3] Arvind, R.S.Nikhil, and K.K.Pingali : "I-Structures: Data Structures for Parallel Computing", ACM Trans. on Prog. Lang. and Syst., Vol.11, No.4, pp.598-632, 1989.

[4] L. Augustsson and T. Johnsson : "Parallel Graph Reduction with the $\langle \nu, G \rangle$-machine", ACM Proc. 4th FPCA, pp.202-213, 1988.

[5] D. E. Culler et al. : "Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine" Proc. of 4th ASPLOS, 1991.

[6] D. C. Cann : "The Optimizing SISAL Compiler: Version 12.0".

[7] P. Hudak : "Exploring Parafunctional Programming: Separating the What from the How", IEEE Software, Vol.5, No.1, pp.54-61, 1988.

[8] Sequent Computer Systems,Inc. : *Symmetry System Summary.*

[9] T. Shimizu, T. Horie, and H. Ishihata : "Low-latency message communication support for the AP1000" Proc. 19th ISCA, pp.288-297, 1992.

[10] E. Takahashi et al. : "Compiling Technique Based on Dataflow Analysis for Functional Programming Language *Valid*", Proc. of SISAL'93, pp.47-58, 1993.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)
1994 IFIP.

Dataflow and Logicflow Models for Defining a Parallel Prolog Abstract Machine

P. Kacsuk*

KFKI-MSZKI Research Institute
H-1525 Budapest, P.O.Box 49, HUNGARY
e-mail:kacsuk@emo.mszki.kfki.hu

Abstract: The Generalized Dataflow Model is introduced for OR- and pipeline AND-parallel execution of logic programs. A higher level abstraction of the dataflow model called the Logicflow Model is applied to implement Prolog on massively parallel distributed memory computers. The paper describes the main features of the two models explaining their firing rules and transition functions respectively.

1 Introduction

There have been several attempts to implement Prolog or other logic programming languages on dataflow computers [5], [6], [2], [3]. The main problem with these approaches was that they aimed at a very fine-grain implementation scheme that required too much parallel overhead. Recently two projects aimed at again the parallel implementation of Prolog based on the dataflow principles on either a coarser-grain platfrom [1] or on fine-grain multi-threaded massively parallel computers [4]. In our project called the LOGFLOW project [8] we want to exploit coarse-grain parallelism based on the data driven semantics in order to implement Prolog on massively parallel distributed memory multicomputers. The LOGFLOW-2 machine is a two-layer distributed memory multicomputer specialized for execution of logic programs. It has two layers:

1. Sequential Prolog Processor (SPP) layer
2. Intelligent Logic network (ILN) Layer

A Prolog program is a collection of sequential modules which can be executed in parallel based on the OR-parallel and AND-parallel principles. The first layer of LOGFLOW-2 (SPP) is a collection of sequential Prolog processors. Each of them is a Warren Abstract Machine (WAM) to be used for efficiently executing sequential Prolog modules. These WAM processors perform coarse-grain logic programming computations independently of each other. It is the task of the second layer (ILN) to organize the work of these WAM processors into a coherent parallel Prolog system. ILN provides an intelligent communication layer specialized for connecting and organizing the work of the WAM processors according to the OR-parallel and AND-parallel principles. Processors of the ILN layer work in a data driven way based on the Dataflow Search Graph (DSG) of logic programs. The DSG is mapped into this processor layer and applying a dataflow execution scheme, nodes of the DSG perform logic and control functions.

The main objectives and contributions of the paper are:
1. to define the Generalized Dataflow Model for parallel execution of pure logic programs (Section 2)
2. to introduce the Logicflow Model by structuring the dataflow subgraphs of the Generalized Dataflow Model into higher level compound nodes and by generalizing the firing rules (Section 3).

* The current paper is part of the project titled "Highly Parallel Implementation of Prolog on Distributed Memory Computers" supported by the National Scientific Research Foundation (Hungary) under the Grant Number T4045.

2 A Dataflow Execution Model for Pure Logic Programs

In the current paper we use the term *reduced Prolog program (RPP)*, which means a Prolog program without any built-in predicates. It can be shown that a reduced Prolog program can be translated into a dataflow graph containing the following elementary nodes:

E_UNIFY(1,3), BACK_UNIFY(2,2), U_MERGE(2,1), E_UNIT(1,1),
AND_SAVE(1,2), AND_RESTORE(2,2),
OR_SAVE(1,4), OR_RESTORE_L(2,2), OR_RESTORE_R(2,2), OR_MERGE(2,1)

Here the first value in the brackets represents the number of input arcs and the second value the number of output arcs. The E_UNIFY, BACK_UNIFY and U_MERGE nodes are closely related and jointly represent the head of a rule clause (see Fig. 1). The E_UNIT node represents a *unit* clause. The AND_SAVE and AND_RESTORE nodes together can be used to connect a goal to the body of a rule clause (see Fig. 2) and finally the OR_XX nodes together serve to connect two alternative clauses in a predicate (Fig. 3). Based on these subgraphs any reduced Prolog program can be translated into a dataflow graph consisting of the elementary nodes and subgraphs introduced above. Figure 4 shows a simple Prolog program and its dataflow graph clearly indicating the applied subgraphs.

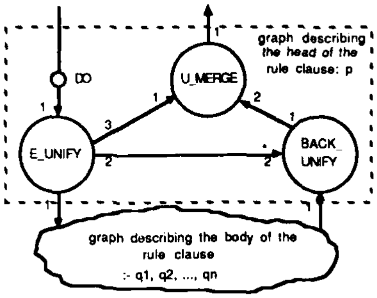

graph describing the head of the rule clause: p
DO
U_MERGE
E_UNIFY
BACK_UNIFY
1
1
2
3
1
2
2
1
graph describing the body of the rule clause :- q1, q2, ..., qn

Figure 1. Dataflow graph representation of rule clause p:-q1,q2,...,qn

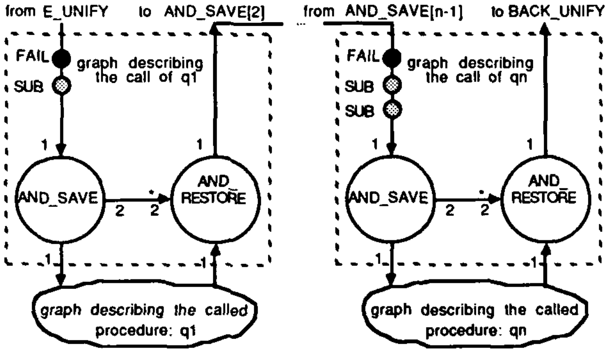

from E_UNIFY
to AND_SAVE[2]
from AND_SAVE[n-1]
to BACK_UNIFY
FAIL
SUB
graph describing the call of q1
AND_SAVE
AND_RESTORE
1
1
2
2
1
1
graph describing the called procedure: q1
FAIL
SUB
SUB
graph describing the call of qn
AND_SAVE
AND_RESTORE
1
1
2
2
1
1
graph describing the called procedure: qn

Figure 2. Dataflow graph representation of the body in clause p:-q1,q2,...,qn

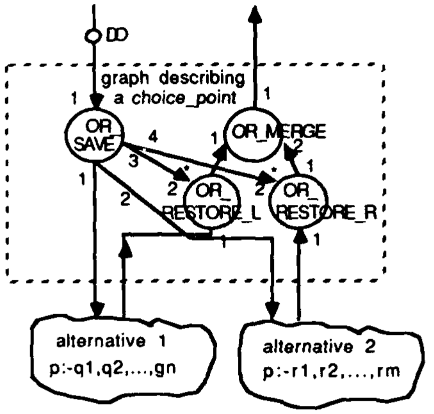


Figure 3. Dataflow graph representation of clause alternatives

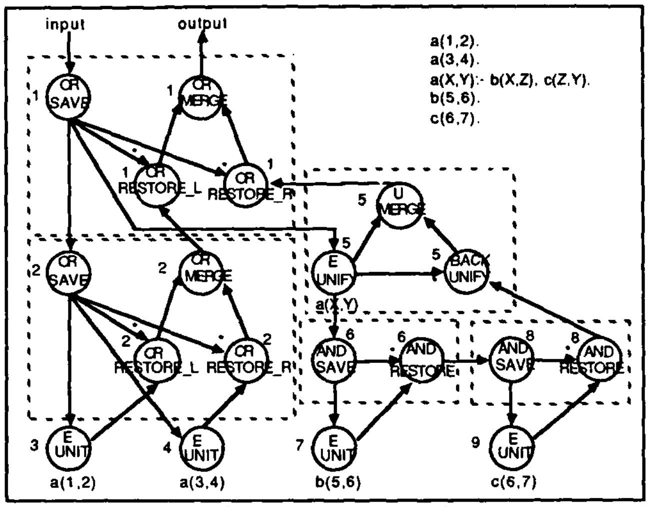


Figure 4. Dataflow graph of a simple Prolog program

The dynamic behaviour of the graph is defined by tokens moving on the arcs and by the firing rules and functions of the nodes. We want to define these features of the graph in such a way that the dynamic execution scheme would result in an OR-parallel and pipeline AND-parallel pure logic programming execution mechanism. We use the term pure logic programming instead of Prolog since the order of executing alternative clauses (i.e. the search strategy) is fixed in Prolog, while in a pure logic programming system this order can be freely selected. In an OR-parallel execution scheme we want to permit any order of execution to exploit as much parallelism as possible. In order to fulfil this task we have generalized the pure dataflow computational model in several points. The new model is called the Generalized Dataflow Model (GDM). In the GDM a node is defined by a 3-tuple:

$\{ \Phi, \Lambda, A \}$

where Φ represents the firing rules of the node
Λ represents the set of logic functions of the node
A represents the set of arguments stored in the node

The main features of the GDM are as follows:

a. There can be nodes which are able to fire even if only a subset of their input arcs contain tokens.
b. There can be nodes which have different firing rules depending on the types of input tokens or on the result of the function executed by the node.
c. There can be arcs for which the node puts back the consumed token after firing. These arcs are called repeat_arcs and are denoted by a "*" on figures.
d. Tokens have context field which is a 3-tuple containing the color, target and return of the token.
e. There can be nodes which generate a token stream on an output arc. All the tokens of a token stream have the same context.
f. There can be nodes which are able to store constant values as arguments for their associated logic function.

Five token types are introduced into the dataflow execution model to describe the activities of a pure logic programming system. The role of these tokens are as follows:

1. request token representing a query:	DO (context,environment,arguments)
2. reply token representing a successful resolution:	SUCC (context,environment)
3. reply token representing a failed resolution:	FAIL (context)
4. request/reply token representing variable substitutions within the clause body:	SUB(context,environment)
5. context token storing temporarily the context of a request DO token:	CONT(context) or CONT(context,environment)

The context field contains the color of the token the target node and arc of the token and a return field identifying the node and arc where the reply token should be sent back. The environment field contains the permanent variables (with their binding values) of the clause in which the current token is moving. The argument field contains the arguments of goal represented by a DO token. The computation is based on the concept of token streams. There are two kinds of token streams:

(a) In reply to a request DO token each node produces a SUCC token stream, which is a series of tokens consisting of N consecutive SUCC tokens closed by one terminating FAIL token. The empty stream consists of a single FAIL token.

(b) UNIFY and AND nodes produce SUB token streams, which consist of N consecutive SUB tokens ended by one FAIL token. The empty stream consists of a single FAIL token.

With these generalizations the firing rules of the elementary nodes are given by the following notation:

(inp_tok,logic_function) => out_tok

where "inp_tok" means the set of input arcs and input tokens taking part in the firing of the node. "logic function" means a logic related function to be executed by the node during the firing. "out_tok" represents the set of output tokens produced by the firing. The use of different kinds of brackets in the definition of firing rules is summarized below:

[] used for token streams, i.e. for tokens to be created sequentially
{ } used for tokens to be available or to be created at the same time
<> used for describing token contexts

Nodes are identified by their type and by an integer as their index number. Notice that a unique index would be sufficient for this purpose, however for the sake of readability of the firing rules we introduced this redundancy into the notation. Accordingly, the closely related elementary nodes (see Fig. 1.-3.) have the same index number. Explanation of some further notations are given here:

i: represents the index of the sender node
k: represents the index of the current node
m or **n**: represent the index of the target node in a token generated by the current node
θ: represents the most general unifier
$E_i\theta_k$: represents the caller environment with the binding values created by the current unification
$E_k\theta_k$: represents the current clause environment with the binding values created by the current unification

Based on these notations the definitions of the elementary nodes are given both formally and informally. A detailed BNF description of the firing rules can be found in [9]. Note that arcs are identified only by numbers in the figures, since the direction of an arc indicates if the arc is input or output.

2.1 Representation of Rule Clause Heads

Defining the E_UNIFY, BACK_UNIFY and U_MERGE nodes we should understand that these nodes are closely related, together they represent the head of a rule clause as shown in Fig. 1.

E_UNIFY:
The task of E_UNIFY is to execute unification between the arguments of the input DO token and the arguments of the represented clause head. Notice that the latter arguments are stored in the E_UNIFY node as constant parameters.

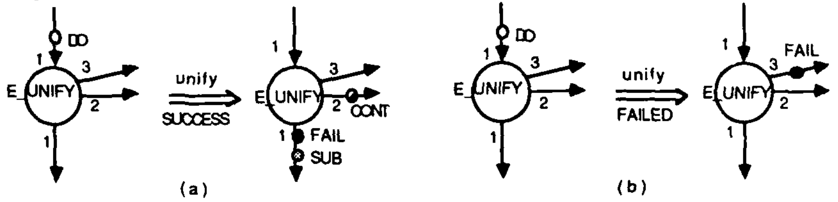


Figure 5. Work of the E_UNIFY node

a/ In the case of successful unification a SUB token stream is generated on arc out1 and a CONT token with the same unique new color on arc out2. The SUB token environment field contains the environment (permanent variables) of the rule clause with the binding values created during the unification. The target node of the stream should be the arc in1 of an AND_SAVE node. The context and the environment field (with the bindings created during the unification) of the consumed DO token are saved into the CONT token which is forwarded to the coupled BACK_UNIFY node.
FR1:

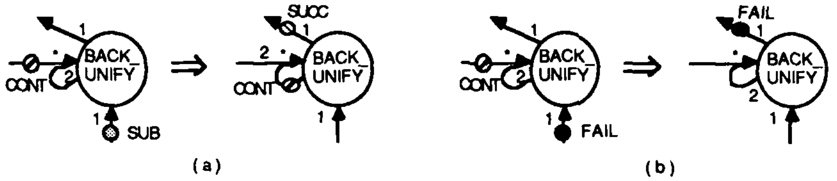
$$\mathbf{(token[in1]=DO(C_i,E_i,A_i),\ unify(E_i,A_i,E_k,A_k)=<E_i\theta_k,E_k\theta_k>)\ =>}$$
$$\mathbf{\{\ token[out1]:=[SUB(C_k,E_k\theta_k),\ FAIL(C_k)],}$$
$$\mathbf{token[out2]:=CONT(<c_k,(BACK\_UNIFY[k],2),C_i>,E_i\theta_k)\ \}}$$
$$\mathbf{where\quad C_i = <c_i,(E\_UNIFY[k],1),r_i>}$$
$$\mathbf{C_k = <c_k,(AND\_SAVE[m],1),\_>}$$

b/ Failed unification (function=unify, result=FAILED): If the unification is failed a FAIL token is created and placed on arc out3. This token inherits the color and return field of the context of the DO token.
FR2:

$$\mathbf{(token[in1]=DO(<c_i,(E\_UNIFY[k],1),r_i>,E_i,A_i),\ unify(E_i.A_i,E_k,A_k)=FAILED)\quad =>}$$
$$\mathbf{token[out3]:=FAIL(<c_i,(U\_MERGE[k],1),r_i>)}$$

BACK_UNIFY:
a/ When SUB and CONT tokens with the same color are available on the input arcs, BACK_UNIFY consumes

them and executes back-unification between the environment fields of the two consumed tokens. The result is placed in a SUCC token on arc out1 which is connected to arc in2 of the coupled U_MERGE node. The saved color of the caller DO token (consumed by the coupled E_UNIFY node) is restored in the SUCC token based on the return field of the CONT token. After firing the node puts back the CONT token on arc in2 as shown in Fig. 6.a.



(a) (b)

Figure 6. Work of the BACK_UNIFY node

FR3:
({token[in1]=SUB(<c_k,(BACK_UNIFY[k],1),_>,E_k),
token[in2]=CONT(<c_k,(BACK_UNIFY[k],2),C_i>,$E_i\theta_k$)[2]},
back_unify($E_i\theta_k$,E_k)=E_i') =>
{token[out1]:=SUCC(<c_i,(U_MERGE[k],2),r_i>,E_i'),
token[out2]:=CONT(<c_k,(BACK_UNIFY[k],2),C_i>,$E_i\theta_k$) }
where C_i = <c_i,(E_UNIFY[k],1),r_i>

b/ When a FAIL token arrives at arc in1 both the FAIL and the CONT tokens having the same color are consumed and a FAIL token is placed on the arc out1. The context of the FAIL token is composed in the same way as the context of the SUCC token.

FR4:
({token[in1]=*FAIL(<c_k,(BACK_UNIFY[k],1), _>)*,
token[in2]=CONT(<c_k,(BACK_UNIFY[k],2),C_i>,$E_i\theta_k$)}, _) =>
token[out1]:=FAIL(<c_i,(U_MERGE[k],2),r_i>)
where C_i = <c_i,(E_UNIFY[k],1),r_i>

U_MERGE:
No matter which token arrives at which input arc, U_MERGE consumes the token and copies it on the output arc. However, the context of the copied token is modified, the target field is inherited from the return field of the consumed token.

FR5, FR6, FR7:
(token[in1]=FAIL(<c_i,(U_MERGE[k],1),r_i>), _) => token[out1]:=FAIL(<c_i,r_i,_>)
(token[in2]=FAIL(<c_i,(U_MERGE[k],2),r_i>)>, _) => token[out1]:=FAIL(<c_i,r_i,_>)
(token[in2]=SUCC(<c_i,(U_MERGE[k],2),r_i>,E_i'),_) => token[out1]:=SUCC(<c_i,r_i,_>,E_i')

2.2 Representation of Unit Clauses

E_UNIT:
The E_UNIT node represents a unit clause. Its task is to execute unification between the input DO token's arguments and the head arguments which are constant parameters of the unify function.

a/ Successful unification: In this case a SUCC token stream is generated on the output arc. The SUCC token's environment field contains the environment of the DO token updated with the binding values created during the unification. The color of the token stream is identical to the color of the DO token, while the target field of the token stream is inherited from the return field of the DO token.

FR8:
(token[in1]=DO(<c_i,(E_UNIT[k],1),r_i>,E_i,A_i),unify(E_i,A_i,E_k,A_k)=<$E_i\theta_k$,$E_k\theta_k$>) =>
token[out1]:=[SUCC(<c_i,r_i,_>,$E_i\theta_k$), FAIL(<c_i,r_i,_>)]

b/ If the unification fails, a FAIL token is placed on the output arc. The target field of the FAIL token is inherited from the return field of the DO token.

FR9:
(token[in1]=DO(<c_i,(E_UNIT[k],1),r_i>,E_i,A_i), unify(E_i,A_i,E_k,A_k)=FAILED) =>
token[out1]:=FAIL(<c_i,r_i,_>)

2.3 Representation of Body Goals

The AND_SAVE and AND_RESTORE nodes are closely coupled. A pair of them is used together for preparing the call of a rule body goal as shown in Figure 2. The task of a connected (AND_SAVE, AND_RESTORE) pair is to receive an input SUB token stream and to produce an output SUB token stream. From each SUB token of the input stream a DO token is generated and sent to the graph representing the procedure to be called by the related body goal. The called graph can create a SUCC token stream for each DO token. The AND_RESTORE node should collect these streams and merge them into a single output stream.

AND_SAVE:
a/ In the case of an input SUB token the node creates a DO token on arc out1 and a CONT token on arc out2. The DO token contains in its argument field the arguments of the corresponding goal. The color of the DO and CONT tokens is inherited from the SUB token.

FR10:
(token[in1]=SUB(C_i,E_i), create_arg(E_i,A_k)=A_k') =>
{token[out1]:=DO(<c_i,$node_m$,(AND_RESTORE[k],1)>,E_i,A_k'),
token[out2]:=CONT(<c_i,(AND_RESTORE[k],2),_>) }
where C_i = <c_i,(AND_SAVE[k],1),_>
$node_m$ = (OR_SAVE[m],1) or (E_UNIFY[m],1)

b/ If a FAIL token arrives, the node simply copies it to the arc out2.

FR11:
(token[in1]=FAIL(<c_i,(AND_SAVE[k],1),_>)[1],_) =>
token[out2]:=FAIL(<c_i,(AND_RESTORE[k],2),_>)

AND_RESTORE:
a/ Whenever a SUCC token arrives from the graph describing the called procedure a CONT token with the same color should be available on arc in2. At this time the AND_RESTORE node can fire. It creates a SUB token on arc out1 based on the context of the consumed CONT token and the environment field of the consumed SUCC token. The CONT token is placed back to arc[2] and therefore it will be available again on arc[2] when another SUCC token belonging to the same reply stream arrives.

FR12:
({token[in2]=CONT(<c_i,(AND_RESTORE[k],2),_>),
token[in1]=SUCC(<c_i,(AND_RESTORE[k],1),_>,E_i) }, _) =>
{token[out2]:=CONT(<c_i,(AND_RESTORE[k],2),_>),
token[out1]:=SUB(<c_i,$node_n$,_>,E_i) }
where $node_n$ = (AND_SAVE[m],1) or (BACK_UNIFY[i],1)

b/ When the stream ending FAIL token arrives at arc in1, both the FAIL token and the CONT token with the same color are consumed and no token is generated. This is the way how an input SUCC stream is transfered into SUB tokens without a stream ending FAIL token.

FR13:
({token[in2]=CONT(<c_i,(AND_RESTORE[k],2),_>),
token[in1]=FAIL(<c_i,(AND_RESTORE[k],1),_>) }, _) => {}

c/ Merging of the input SUCC token streams with the same color is finished when all the CONT tokens having the same color has been consumed from arc[2]. If there is a FAIL token here and no CONT token with the same color, the FAIL token is simply copied on arc out1 and in this way the output SUB tokens are supplied with a stream ending FAIL token. Notice that the FAIL token on arc in2 must no: be consumed as long as there is any CONT token with the same color on that arc.

FR14:

```
(token[in2]=FAIL(<$c_i$,(AND_RESTORE[k],2),_>), _)   =>
                         token[out1j:=FAIL(<$c_i$,$node_n$,_>)
wher:     $node_n$ = (AND_SAVE[m],1) or (BACK_UNIFY[i],1)
```

2.4 Connection of Alternative Clauses

The OR_SAVE, OR_RESTORE and OR_MERGE nodes are closely related too. They serve to connect alternative clauses of a predicate, i.e. to realize choice points in the Prolog Search Tree as shown in Figure 3. Due to the lack of space here we give only the firing rule of the OR_SAVE node, the others are described in [9].

OR_SAVE:
Copies the input DO token onto arcs out1 and out2 with a new color. The context of the DO token is saved into two CONT tokens.

FR15:

```
(token[in1]=DO($C_i$,$E_i$,$A_i$), _)      =>
                  { token[out1]:=DO($C_{kl}$,$E_i$,$A_i$),  token[out2]:=DO($C_{kr}$,$E_i$,$A_i$),
                  token[out3]:=CONT(<$c_k$,(OR_RESTORE_L[k],2),$C_i$>),
                  token[out4]:=CONT(<$c_k$,(OR_RESTORE_R[k],2),$C_i$>) }
where     $C_i$ = <$c_i$,(OR_SAVE[k],1),$r_i$>
          $C_{kl}$ = <$c_k$,$node_{left}$,(OR_RESTORE_L[k],1)>
          $C_{kr}$ = <$c_k$,$node_{right}$,(OR_RESTORE_R[k],1)>
          $node_{left}$ is the first node of the first alternative clause
          $node_{right}$ is the first node of the second alternative clause
```

3 LOGICFLOW: A Higher Level Abstraction

Based on the node definitions given in Section 2 we can transform pure logic programs into dataflow graphs and the dataflow execution model results in an OR-parallel and pipeline AND-parallel, all-solution Prolog system, where possible solutions for a query are delivered by a SUCC token stream (see proof in [9]). However, there are several motivations to increase the granularity of the nodes in the execution model:

1. The dataflow model is very fine-grain and consequently we can expect large overhead in an actual implementation.
2. The dataflow graph is very unstructured, it is difficult to see how Prolog programs are transformed into the graph.
3. Even for small logic programs large dataflow graphs would be generated.
4. Our purpose is to implement Prolog on coarse-grain distributed memory multicomputers, where the communication overhead is even higher than in a dataflow machine.

In order to increase the granularity of the execution model we introduce a higher level of abstraction where the closely related elementary nodes are grouped together and for these groups new node types are introduced. These new node types are as follows:

UNIFY:	for executing unification on rule clause heads and entering binding results into clause bodies
AND:	for connecting body goals
OR:	for connecting alternative clauses of a predicate

The graphical notation of the new nodes and their relation to the elementary nodes can be seen in Fig. 7. The new compound nodes are even farther from the original pure dataflow concept than the elementary nodes. In order to describe these nodes we should further generalize the concept of dataflow and additionally to properties a/-f/ described in Section 2 we should introduce a new one:

g. *Nodes can have inner state, which can be modified during the firing of the node.*

The new model is called the Logicflow Model. Generally saying a node is defined by a 4-tuple in logicflow:

(Φ, Λ, A, Σ)

where Φ represents the set of transition functions of the node
Λ represents the set of logic functions of the node
A represents the set of arguments stored in the node
Σ represents the set of states of the node.

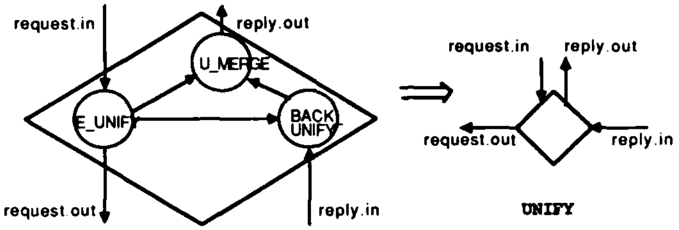


Figure 7.a Structure and graphical notation of UNIFY node

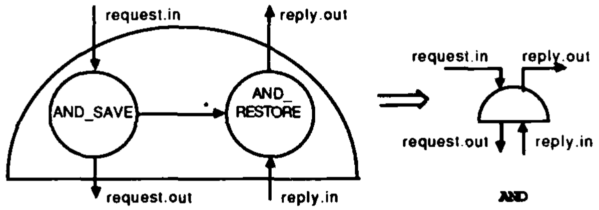


Figure 7.b Structure and graphical notation of AND node

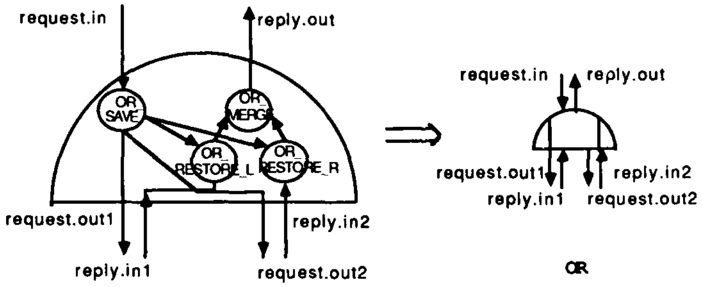


Figure 7.c Structure and graphical notation of OR node

Notice that the CONT tokens of the GDM become hidden tokens in the Logicflow Model, since these tokens are invisible outside the nodes. In order to maintain these tokens we need a new representation form which is called the inner state of nodes. The inner state represents the contents of the hidden CONT tokens. Since the inner state of a node has influence on the firing rule of the node we have to generalize the firing rule of the dataflow model. The generalized firing rule is called as the transition function. The description format of the transition functions for the Logicflow Model is as follows:

(state1, inp_tok, logic_function) => (state2, out_tok)

The transition functions of the compound nodes can easily be derived from the firing rules of the component elementary nodes. As an example firing rules of the UNIFY node are given in this paper, detailed explanation and the transition functions of other compound nodes can be found in [9].

TF1:

(state=_, token[request.in]=DO(C_i,E_i,A_i), unify(E_i,A_i,E_k,A_k)=FAILED) =>
(state:=_, token[reply.out]:=FAIL(<c_i,r_i,_>))
where *C_i = <c_i,(UNIFY[k],request.in),r_i>*

TF2:

(state=_, token[request.in]=DO(C_i,E_i,A_i), unify(E_i,A_i,E_k,A_k)=<$E_i\theta_k,E_k\theta_k$>) =>
(state[c_k]:=<$c_i,r_i,E_i\theta_k$>, token[request.out]:=[SUB($C_k,E_k\theta_k$), FAIL(C_k)])
where C_i = <c_i,(UNIFY[k],request.in),r_i>
C_k = <c_k,(AND[m],request.in),_>

TF3:

(state[c_k]=<$c_i,r_i,E_i\theta_k$>, token[reply.in]=SUB(<c_k,(UNIFY[k],request.in),_>,E_k),
back_unify($E_i\theta_k,E_k$)=E_i') =>
(state[c_k]:=<$c_i,r_i,E_i\theta_k$>,token[reply.out]:=SUCC(<c_i,r_i,_>,E_i')>

TF4:

(state[c_k]=<$c_i,r_i,E_i\theta_k$>, token[reply.in]=FAIL(<c_k,(UNIFY[k],reply.in),_>),_) =>
(state:=_, token[reply.out]:=FAIL(<c_i,r_i,_>))

4 Summary and Conclusions

The introduced Generalized Dataflow Model constitutes a theoretical basis for defining a parallel Prolog abstract machine. Further generalisation leads to the Logicflow Model where compound nodes are defined as a network of elementary nodes and hidden tokens inside the compound nodes are realized by inner state (or memory). In the Logicflow Model compound nodes can fire even if only a single input arc contains a token. This property of the model can advantageosly be exploited by avoiding the application of costly and slow matching stores in the underlying parallel architecture. Based on the Logicflow Model the Distributed Data Driven Prolog Abstract Machine (3DPAM) has been defined in [7].

References

[1] Baiardi, F., Candelieri, A. and Ricci, L. *A Data-Driven Static Model for the Execution of Logic programs on Distributed Memory Systems*, Proc. of JICSLP'92 Post-Conf. Joint Workshop on Distr. and Parallel Impl. of Logic Prog. Sys., 1992

[2] Bic,L and Lee,C. *A Data-Driven Model for a Subset of Logic Programming*, ACM Trans. on Prog. Lang. Syst., vol. 9, no. 4. 1987, 618-645

[3] Biswas, P. and Tseng C. *LogDf: A Data-Driven Abstract Machine Model for Parallel Execution of Logic Programs*, Proc. of the Int. Conf. on Fifth Gen. Comp. Sys., 1988, 1059-1070

[4] Gaudiot, J-L, et al. *Data-Driven Execution of Logic Languages*, Proc. of ICLP'93 Post-Conf. Workshop on Concurrent, Distributed and Parallel Implementation of Logic Prog. Systems, Budapest, 1993

[5] Hasegawa,R. and Amamiya,M. *Parallel Execution of Logic Programs based on Dataflow*, Proc. of the ICOT Conference 1984, 507-516

[6] Halim,Z. *A Data Driven Machine for OR-Parallel Evaluation of Logic Programs*, New Generation Computing, 1986, 5-33

[7] Kacsuk,P. *Distributed Data Driven Prolog Abstract Machine*, in Implementations of Distributed Prolog, Edited by P. Kacsuk and M.J.Wise, John Wiley,1992, 89-118

[8] Kacsuk,P. *LOGFLOW-2: A Transputer Based Data Driven Parallel Prolog Machine*, in Proc. of the World Transputer Congress'93, Aachen, 1993, 1154-1169

[9] Kacsuk,P. *Execution Models for Parallel Implementation of Prolog*, in Proc. of the Int. Workshop on Computational Models and their Applications, Antalya, 1994

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Towards a Computational Model for UFO

John Sargeant, Chris Kirkham, Steve Anderson

Dept. of Computer Science, University of Manchester, Manchester M13 9PL, UK
{js,cck,andersos}@cs.man.ac.uk

Abstract: This paper motivates and outlines a general-purpose, architecture independent, parallel computational model, which captures the intuitions which underly the design of the United Functions and Objects programming language.

The model has two aspects, which turn out to be a traditional dataflow model and an actor-like model, with a very simple interface between the two. Certain aspects of the model, particularly strictness, maximum parallelism, and lack of suspension are stressed. The implications of introducing stateful objects are carefully spelled out.

The model has several purposes, although we largely describe it as it would be used for visualising the execution of programs.

Keyword Codes: D.3.3
Keywords: *Language Constructs and Features*

1 Introduction

We seek a computational model for general-purpose parallelism. Such a model would provide both a static, architecture-independent, representation for parallel programs, to be used by compilers and a dynamic, architecture-independent, representation for such programs, to be used by visualisation, debugging, and profiling tools, and could form the basis of a real high-performance parallel implementation, on suitable architectures.

This paper concentrates on the motivation for, and main ideas of, the proposed model. Many details, in particular the graphical representation of the model, are omitted due to lack of space, and the interested reader is encouraged to ftp [1].

1.1 Properties required of the model

While the above aims are clearly aggressive, we believe they are possible, despite the numerous failed attempts which litter the history of the subject through the 80s. However they impose stringent constraints on the form of the model.

1. It should be intuitively natural, so that people can use tools based on it.

2. It should allow the expression of parallelism at any level of granularity.

3. It should allow expression of locality and modularity in parallel computation. In particular it should allow local synchronisation to be expressed naturally.

4. It should be capable of efficient implementation, at least in principle, either directly on suitable specialised architectures, or by transformation to code appropriate for conventional machines.

5. It should have a clean mathematical description and be amenable, at least in principle, to formal reasoning.

Although, for pragmatic reasons, we place a relatively low priority on formal reasoning, we believe that the model described below should be amenable to such reasoning. The stress on intuitiveness implies that the model cannot be totally language-independent; the programmer must be able to relate what is going on to the source program. The design of such a model therefore reflects a set of choices about what is the "right" sort of programming language for general-purpose parallel computation. Our choices are embodied in the United Functions and Objects programming language, whose main characteristics are summarised below.

1.2 Models considered unsuitable

Many models can be eliminated if these principles are accepted. We would first propose to eliminate any model based on communicating sequential threads(CST). This is clearly a radical and controversial step, as most existing parallel computation is expressed in terms of such threads. In our view, the most fundamental question in this whole field is whether the best way to view parallel computation is in terms of such threads, or in terms of some more radical, more "totally parallel" model.

We would argue against CST on a number of grounds, most of which stem from the fact that a CST program contains discontinuities between the insides of threads, which execute serially, and the communication between them, which is where the parallelism is. The placings of these discontinuities is, in practice, often arbitrary, and changing them may change the semantics of the program. Threads may be related to objects in a concurrent OOP style, but this is at best a marriage of convenience: we see no compelling reason why the unit of encapsulation should also be the boundary between serial and parallel execution. Indeed, the whole notion of a "thread of control" 's an artificial one, an artifact of the way serial computers work rather than a property which is readily identifiable in real-world objects. Note that we are not ruling out active objects - ones which do autonomuous computation rather than merely reacting to incoming messages - but we are questioning the assumption that such autonomous computation should be equated to a thread of control.

The best size for a thread is machine-specific and may vary widely; clearly this makes it undesirable for the programmer to determine the thread boundaries explicitly. On the other hand, if the compiler does it, there is a problem mapping the partitioned code back into the form the programmer knows about. If thread sizes are statically determined, poor performance results for irregular programs; on the other hand use of the techniques required for such programs, such as dynamic granularity and lazy task creation[11] makes the dynamic behaviour very complex. For instance[3] shows how the behaviour of a very simple program under such a model can vary dramatically with subtle changes in the exact details of how threads are spawned.

Our aim, then, is to find the best alternative to CST, although on conventional machines, we may well implement out model using CST techniques.

Graph reduction, and graph rewriting techniques in general suffer from well known efficiency problems, but in our view the lack of intuitiveness (principle 1) is a more fundamental problem. For straightforward strict functional computation, graph reduction is a reasonable representation, but there are simpler ones. For more general computations (lazy or stateful), graph rewriting provides a very general representation, but the graph transformations are low-level and difficult to visualise and, in the case of laziness, the execution order is even more so.

1.3 So what is left?

We only know of two existing models which may meet the criteria, namely the **dataflow** model and the **actor** model.

We do not have a completely new model to propose, and we would be surprised to discover a completely new model which meets the above constraints, so we are left with these two possibilities, or some combination. Before we explore this further, we need to briefly describe the key ideas of the UFO language.

1.4 United Functions and Objects

Space permits only a listing of the key ideas here. Readers wishing to learn about the UFO language in detail should consult the UFO papers [2, 7]. The relevant features of UFO are:

- Parallelism by default; sequencing is by data dependence, and by queueing for access to stateful objects, only. This implies that the computational model is *not allowed* to hide parallelism and also that it needs local synchronisation.
- A well-defined subset of UFO is a higher-order functional language, but incorporating the OO notions of classes and inheritance with dynamic binding. It also has integrated functional arrays and loop structures, in the style of SISAL [6].
- UFO has stateful objects, with updateable instance variables, which are accessed by a queuing mechanism like actors [5]. A single-update scheme provides internal coherence. Also like some actor languages, UFO provides conditional message acceptance. An object may accept certain messages only when particular conditions (defined in terms of the values of the instance variables) are met. Amongst other things, this allows non-strict or lazy data structures to be programmed if required.
- Unlike most actor languages, e.g. ABCL[15] and HAL[8], UFO has a static type system which provides type safety but with considerable flexibility. Overloading, subtyping via inheritance, and genericity are all supported. In addition, the type system provides important static protection from some of the problems associated with the transition from a pure functional environment. In particular, stateful objects can always be identified statically, and there is a static guarantee that a function cannot modify its arguments, even if they are stateful.[1].

[1] The guarantee is a structural one, not merely a parameter passing convention

2 The UFLOW model

2.1 Dataflow or actors?

The UFO language design, and the motivation for that design, strongly suggest that aspects of both models are required. Dataflow is the obvious way to represent the (strict) functional computation which is the natural way to write many applications (especially numerical ones) in UFO. On the other hand, UFO provides the facilities of a concurrent object-oriented language, and the natural way to view such computation is in the actor style. The duality is illustrated by the syntax of a method call, where the functional style `f(x,y)` and the OO-style `x.f(y)` (in OO-speak: send message `f(y)` to the object `x`) are interchangeable.

In the dataflow model, data, as tokens on arcs, flows to methods at nodes. In the actor model, messages containing methods flow to nodes containing objects. So the models are in some sense inverses of each other. The other significant difference is that, in the simplest versions at least, dataflow describes purely functional computation, while actors have state.

We considered two possible approaches. The first attempted to give a single unified view of the computation, allowing both aspects to be represented in the same graph structure. It turns out that this can be done, but the resulting model is rather complex and unsatisfactory in a number of ways. There are several different sorts of nodes and arcs, and the same piece of code can be represented in multiple ways.

We therefore concluded that it was better to have two separate views, the ***computational view*** and the ***object view***. The latter is a straightforward representation of (possibly stateful) objects, and it is mostly the computational view which is of interest here.

The computational view turns out to be a remarkably conventional dataflow model, with strong similarities to IF1[4]. Arcs carry values which are primitive data values, references to objects, or references to methods. A node consists of a number of *slots*, which hold constants or are the targets of input arcs. The first two slots are distinguished. The first slot, when filled, contains (a reference to) the object, i.e. the first (primary) argument of the method. The second slot, when filled, contains the method. The other slots, if any, are for further arguments. e.g. the body of the function:

```
g( x: Int ): Int is { y = 2; return x.f(y) }
```

is translated to:

```
1 S (  C 2   C const ) I  y
2 S (  P x   C Int|f   N 1 ) I
```

Node 1 has two slots, the constant 2 and the "method" `const` which simply outputs the constant value, giving y. Node 2 has 3 slots. `P x` denotes a slot whose input is a parameter `x`, i.e. an input to the graph. The third slot, on the other hand, takes its input y from node 1. In the middle, the method, (which is constant, hence the `C`) is given, using its full disambiguated name `Int|f`. Both nodes are `S`imple, and both have result type `I`nt.

Normally, a node is enabled when all its slots are full. An exception to this occurs when the method call blocks on access to a stateful object (see later). A node, once enabled, will be fired, but not necessarily immediately.[2]

[2] The form of the nodes may suggest a similarity to packet based graph reduction, and therefore inefficiency. However, in this case the process is all data driven, and we do not attempt to (directly) implement non-strict semantics.

The model also includes compound nodes, which represent control structures such as conditionals and loops. These are rather similar to those in IF1, so we omit the description here and once more refer the reader to [1].

2.2 Method call and strictness

In any dataflow model, a key issue is how to model reentrancy. There are two common approaches, namely tagged tokens and graph copying.

Tagged tokens are most appropriate when parts of a graph can start executing before other inputs are available. However, this is inefficient in space, which is particularly problematic for visualisation purposes, and can be a problem in a real implementation, exascerbating the problem of parallelism control or "throttling" [12]. We prefer a model in which a method call happens at an identifiable moment in time, after all the inputs are available, at which point the graph is expanded, and input tokens are placed on the appropriate arcs. Because the method may be dynamically bound, the graph produced may depend on the first element of the node (the object) as well as the second (the method). However, note that this is the *only* way in which dynamic binding affects the model, and other dynamic binding strategies, such as multimethods, could be easily accommodated.

Once the graph has been generated, the node has done its job. The node is conceptually replaced by a *box* containing the graph. The box has no computational role, it merely serves as a container for the nodes. For visualisation purposes, the box remains in place until all activity within it has finished.

We can now summarise some important aspects of the model:

- Data driven: Data flows to nodes which become enabled when all the necessary inputs are present.

- Strictness: all arguments to a node are evaluated before the node is enabled. This may seem restrictive, but it makes the other desirable properties possible.

- Maximum parallelism: Enabled nodes may be fired in any order, but all must be fired eventually.

- No suspended computation. Nodes become enabled, fire, and always terminate. They never suspend or block.

The model as defined thus far has the futher useful property that it is possible to measure the "inherent" parallelism of any program / data set, independently of the target architecture, and independently of the actual evaluation order.

2.3 Higher-order functions

Function values may be represented simply by having references to functions passed on arcs rather than fixed at nodes. Partial application is represented using a node which has dummy slots which do not need to be filled before it fires. The result is a partial application packet, which can then be passed to other nodes, e.g. an alternative representation for `x.f(y)` is to represent the "message" `f(y)` as a partial application:

```
2 S (  _  C Int|f N 1 ) F ( I  ) I
3 S (  P x  N 2 ) I
```

The first slot of node 2 is empty, indicating that no input value is expected. The result of node 2 is the message, a function of type `Int -> Int`, which is directed to the method slot of node 3.

We therefore have messages as first class values "for free". Note that the partial application packet is inserted in a single slot in the node in the usual way; its slots are not "matched" against the slots in the node. Such matching is an attractive idea, but it does not work because a node representing a function application cannot in general "know" whether the incoming function has already been partially applied.

2.4 Parallelism measurement and atomicity

The model as defined thus far has two useful properties which we are about to destroy . the next section. They are still worth highlighting, as many applications, or large sections of applications, can be sensibly written in the (statically distinguished) functional subset of UFO which the simple model describes.

Atomicity

The discussion so far has assumed that graphs are expanded down to atomic nodes. In fact, in any program which terminates and does not crash, we can regard any node as atomic in the sense that we can fire it and it will eventually terminate and produce its result, independently of all other nodes. This is important for visualisation purposes, as it means that the graph can be viewed at any level of granularity, and only those nodes which are of interest need be expanded. At the implementation level, it means that any node may be executed sequentially. From the point of view of measuring parallelism, it allows experimentation with the effects of granularity on parallelism, if we assume that nodes deemed atomic are executed sequentially.

3 Stateful objects and non-strictness

As explained above, stateful objects have mutable state in the form of instance variables. Internal coherence is ensured by allowing a procedure to define exactly one new value for each variable which it updates; the old and new values are syntactically distinguished, like old and new loop values in single-assignment loops. Interference between method calls is avoided by locking an object while a procedure is updating it. Messages queue for access, by default FIFO, but conditional message acceptance can be used to override this; when the object is unlocked, the first acceptable message in the queue is processed next.

Although here we use stateful objects only to implement non-strict computation, we expect that in general they will be used in many application areas. In particular, they may often provide a more natural way of modelling real-world objects than a pure functional representation. Real world objects have identity and state. They retain their identity when their state changes, a property which a pure functional representation does not address.[3]

A simple implementation of non-strictness is enough to sketch the model and highlight the issues, however. The following class implements a straightforward I-Structure location.

[3] A good exposition of this view of the world can be found in [9]. It should be remembered, though, that programs frequently manipulate abstract entities, which are often immutable, as well as real-world objects.

An I structure is one each of whose elements is written exactly once; an attempt may be made to read an element before it has been written, in which case the read is deferred until the write has been made. A presence bit is associated with each location in order to detect whether the write has occurred or not.

```
stateful class ILoc[T]
** Give initial values for the instance variables
{ initial val = null:T;
  initial present = false }

accept read when present

** read is a function
read: T is val

** write is a procedure - note similarity to single-assignment loops.
proc write( v: T ): Void is
 pre
  not present
do
  new val = v;
  new present = true
od
end ** ILoc
```

The use of an array of such objects to implement a wavefront algorithm, and generalisation to more powerful non-strict structures such as futures, are described in [7].

The implementation relies on conditional message acceptance: reads are only accepted after the value is written. In a high performance implementation, this would be a built-in library class, implemented as efficiently as possible, with hardware support if available. [4]

Representation of the methods is straightforward. In particular, the body of procedure write can look exactly like the body of a loop with loop values val and present. The single assignment updates mean that parallelism can be exploited within the body of a procedure just like anywhere else.

For visualising stateful objects, the object view becomes more important. More precisely, encapsulation requires two versions of this view: one of the outside of an object (the state of its message queue, and any public instance values) and one of the inside (the instance variables and private instance values). For instance, an I-Structure with a blocked read would appear in the computational view as a node with all inputs present, but not enabled. In the outside object view, we would see our read request in the queue,

[4] We note in passing that this example neatly illustrates the difference between acceptance conditions and preconditions. If an acceptance condition is not met (read when not present), the method simply is not accepted yet. In the computational view, this appears as a node which, although all the inputs are present, is not yet enabled. If a precondition is not met, (write when present) an error occurs. Also, it seems to us that the acceptance condition is a property of the object: it decides what messages it is currently prepared to accept. The precondition, on the other hand, seems more naturally a property of the method: it decides whether it can cope with the state it finds when it gets there.

poerhaps along with others (the user could follow these back into the computational view). We would also see that the object is is unlocked, but is only accepting writes. In the inside object view, we see the values of the instance variables, and in particular that the presence bit is not set. Typically, a user of the class would not need to see (perhaps even not be allowed to see) the inside view, but the designer of the class would. The object views would also allow the user to follow references between objects.

What are the consequences of introducing state? The maximum parallelism principle still applies: it is still ok to fire enabled nodes in any order provided all are fired eventually. Blocking on stateful objects (either because of mutual exclusion or because of CMA) shows up in the computational view as nodes which are not enabled even though they have all their inputs. There is still no notion of suspended computation. Deadlock appears as a situation where no nodes are enabled, and can be traced by investigating which nodes are blocked and why (by flipping between the views).

However, there are several difficulties. The parallelism in the program is no longer independent of evaluation order: indeed some orders may lead to nontermination or deadlock while others do not. The atomicity property no longer holds in general, as there can be implicit dependencies between nodes which may require one node to fire before another can terminate. In general, programs are much harder to visualise and reason about. Implementation is also more difficult, as described below.

Nevertheless, we believe that stateful objects are necessary for some applications, and that these problems are inherent, and not an artifact of our model. The UFO type system allows us to distinguish statically which objects are stateful. The model provides a natural representation both of the overall computation and of the behaviour of the individual objects.

4 Implementation issues

4.1 Scheduling

The scheduling mechanism must guarantee that all enabled nodes get fired eventually, but otherwise it is not semantically constrained. For visualisation purposes, a user might well decide by clicking a mouse which node to fire next. However, when not user-guided, a scheduling strategy is necessary. The main consideration is limiting the number of nodes in existence at any one time, to limit store usage (and screen usage in the visualisation case). The basic strategy therefore is to go breadth-first until enough parallelism is generated, depth-first thereafter. For loops earlier cycles should be done first. This is straightforward to implement in the pure functional case, but blocking on stateful objects may result in there being no enabled nodes on the "ideal" path.

4.2 Efficiency

Obviously the model maps very directly onto modern dataflow architectures. For instance, the graph expansion on firing a node is equivalent to allocating an operand segment in the EM4 machine.

For more conventional architectures, the code must be transformed to a threads-based model. A suitable model, incorporating variable granularity, lazy task creation, and load balancing via active messages is described in[14]. The main problem with such models is making the threads large enough, as conventional machines tend to have very long (relative to our requirements, at least) context switching times. In the pure functional case, the implementation can use whatever thread granularities it likes (even a single serial thread), and the semantics are preserved. In the presence of stateful objects, this is no longer the case, as there are implicit dependencies between nodes, and using large threads may well cause spurious deadlock.

A naive correct implementation is to generate a separate thread for each node representing a method call on a stateful object. The game then is to optimise this by detecting as many cases as possible where the separate thread is unnecessary, and thus merging threads. The type system provides a certain amount of useful information in this regard.

Of course there are many other issues to address, for instance load balancing and data distribution. For simple applications involving arrays and loops, we expect to be able to apply existing technology such as that used in parallelising Fortran compilers, and particularly that used in the SISAL project[6]. For more irregular computations, we will draw on our own experience with parallel functional implementations [3, 13] and on that from the concurrent OOP community.

4.3 Current status

The intermediate format based on the model has been defined[1], both as a textual format and in terms of a collection of UFO data structures. It provides the basis for a visualisation and debugging tool we are developing, and will be used in a portable optimising UFO compiler for parallel machines, initially targetted at the KSR1 machine at Manchester. Current work is concentrating on sharing and update-in-place analysis, and efficient implementation of stateful objects.

Acknowledgements

The UFO project is supported by a grant from the former Science and Engineering Research Council.

References

[1] J. Sargeant, C. C. Kirkham, S. Anderson: The UFLOW Computational Model and Intermediate Format Technical report UMCS-94-1-5, Department of Computer Science, University of Manchester, 1994.[5]

[2] J. Sargeant: **Uniting Functional and Object-Oriented Programming** Invited Paper, International Sympoisum on Object Technologies for Advanced Software, Kanazawa, Japan, November 1993 LNCS 742, pp 1-26.

[5] UMCS Technical reports are available by anonymous ftp from m1.cs.man.ac.uk, directory /pub/TR, report number as filename.

[3] J. Sargeant, I. Watson: **Some experiments in controlling the dynamic behaviour of parallel functional programs**, Proc. Workshop on the Parallel Implementation of Functional Languages, Southampton, June 1991, Southampton University tech. report CSTR 91-07, pp 103-121.

[4] S. K. Skedzielewski and M. L. Welcome: **Dataflow Graph Optimization in IF1** in J-P. Jouannaud (editor), **Functional Programming Languages and Computer Architecture**, pp17-34, Springer-Verlag, NY, 1985.

[5] G. Agha: **Actors: A Model of Concurrent Computation in Distributed Systems** MIT Press series in artificial intelligence, 1986.

[6] D. C. Cann: **Retire Fortran? A debate rekindled** Communications of the ACM 35(8), August 1992, pp 81-89

[7] J. Sargeant: **United Functions and Objects: An Overview** Technical report UMCS-93-1-4, Department of Computer Science, University of Manchester, 1993.

[8] C. Houck, G. Agha: **HAL: A High-level Actor Language and its Distributed Implementation** Proc. 21st International Conference on Parallel Processing, August 1992.

[9] G. Booch: **Object-Oriented Analysis and Design with applications**, second edition, Benjamin/Cummings 1994.

[10] M. Haines and A. P. W. Böhm: **Towards a Distributed Memory Implementation of Sisal**, Scalable High Performance Computing Conference, April 1992.

[11] E. Mohr, A. Kranz, R.H. Halstead: **Lazy Task Creation: A Tecnique for Increasing the Granularity of Parallel Programs** ACM Conference on Lisp and Functional Programming, Nice, France, June 1990.

[12] C.A. Ruggiero & J. Sargeant, **Control of Parallelism in the Manchester Dataflow Machine**, in Proc. 1987 Conference on Functional Programming Languages and Computer Architecture, Portland, Oregon, 1987.

[13] J. Sargeant: **Improving Compilation of Implicit Parallel Programs by Using Runtime Information**, Proc. Workshop on Compilation of Symbolic Languages for Parallel Computers, San Diego, 1991, Argonne National Laboratory tech. report ANL-91/34, pp129-148.

[14] I. Watson: **SLAM: Lazy Task Creation and Dynamic Load Balancing Using Active Messages.** Internal report, Deptartment of Computer Science University of Manchester 1994. In preparation.

[15] A. Yonezawa (ed.): **ABCL, an Object-oriented Concurrent System,** MIT press Computer Systems Series, 1990.

PART X

SHORT PAPERS

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Software Pipelining: A Genetic Algorithm Approach

V. H. Allan[a] and M.R. O'Neill

[a] Department of Computer Science, Utah State University
Logan, Utah 84322-4205 allanv@vicki.cs.usu.edu

Abstract: *Software pipelining can be facilitated by applying a genetic algorithm to a petri net representation of the cyclic dependencies present in a loop. By restricting the transformations to only increase minimum times between nodes, a legal schedule is always produced. The objective function rewards low effective initiation interval. When compared to the basic petri net algorithm, significant improvements are realized.*
Keyword Codes:C.1.3, D.1.3, D.3.m, I.2.8

1 Software Pipelining and Genetic Algorithms

Software pipelining is a loop optimization technique in which the body of the loop is reformed so an iteration of the loop can start executing before the previous iterations of the loop have finished [RF93]. In this research, we have developed a genetic algorithm for performing software pipelining which is a marked improvement over existing algorithms in result and comprehensiveness.

One can create a dependency graph G(N,A) in which the nodes, N, represent operations and the arcs, A, represent a must follow relationship. An arc $a \rightarrow b$ must be annotated with a *dif,min* pair. *Dif* is the difference in the iterations from which the operations come. *Min* is the time which must elapse between the time the first operation is executed and the time the second operation is executed. Figure 1(b) shows the data dependence graph for the loop body of Figure 1(a). Figure 1(c) shows a dependence constrained schedule for the first three iterations.[1] Iterations progress horizontally while time progresses vertically. Each row in the schedule is a parallel instruction.

The operations shown in the box repeat and form a new loop body. The code before the repeated section is termed the *prelude* while the code after the repeated section is termed the *postlude*. This new loop body together with the prelude and postlude is termed a *software pipeline*. The term *effective initiation interval* is the average number of time units it takes to complete a full iteration. Using petri nets[GWN91] which models data dependency, we are able to create a schedule. Nodes become transitions in the petri net, each arc becomes a *pair* of arcs in the petri net, and places are inserted between dependent transitions to keep track of what may fire next. The number of tokens in each place is determined by the dif value on the arc represented by that place. Resources are added using the tokens within a resource place to model a limit to the number of resources

[1] In this schedule, some operations have been delayed to preserve the regularity of the schedule.

A full copy of this paper is available via anonymous ftp to slow.cs.usu.edu, directory pub/Thesis.

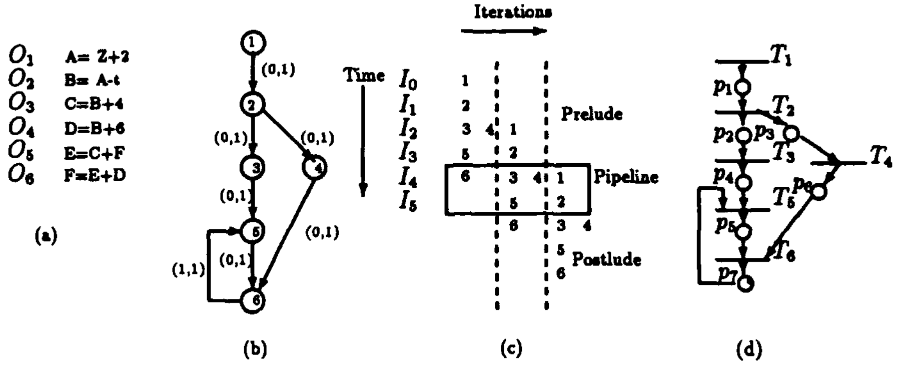
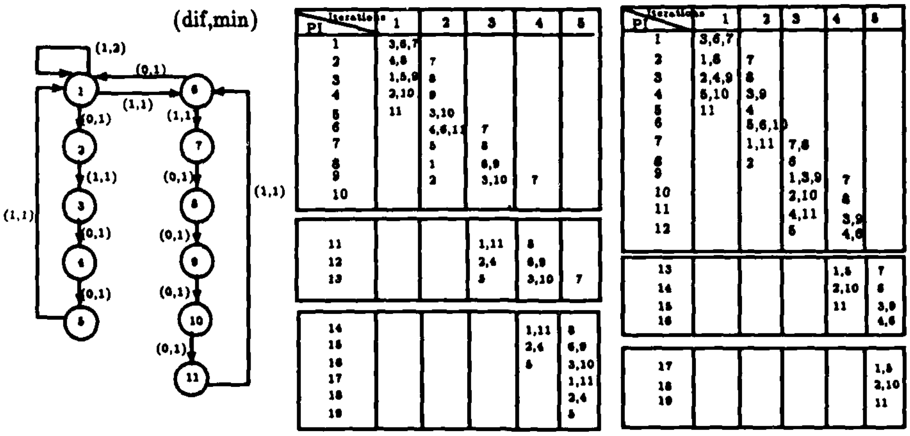


Figure 1: (a) Original loop body (b) Data dependence graph of loop body (c) Software Pipelined loop body (d) Corresponding Petri Net

[RA93]. Since the algorithm is deterministic, it is guaranteed that a repeating pattern will be found. Figure 1(d) shows the corresponding petri net for the example in Figure 1(a).

Using a petri net to generate parallel schedules compares favorably with other techniques [RA93]. Since scheduling is known to be NP-complete, no heuristic technique can hope to produce optimal results. We attempt to produce optimal schedules by coupling the petri net algorithm with a genetic algorithm to locate the best schedule within the search space. Genetic Algorithms (GAs)[Gol89] are stochastic search algorithms that mimic biological genetics in their search for the perfect environmental fit. GAs are based on Darwin's concept of "survival of the fittest" in nature. The algorithm maintains a population of solutions, called individuals or chromosomes, that reproduce for some number of generations. Those individuals that are fittest have the highest probability of contributing to the next generation of individuals. For simplicity, it is assumed that an individual is represented as a fixed string of bits that can be decoded as a solution, and whose fitness can be measured with an objective function (termed the *fitness function*) used to evaluate the quality of the solution. It is up to the user to model the objective function correctly. The GA will attempt to optimize the solutions based on whatever fitness function it is provided. We have chosen to minimize the effective initiation interval because it closely approximates run time, and we can compare our results with Lam[Lam88], who attempts to minimize this metric.

One of the advantages of the genetic algorithm is its ability to effectively search an exponential solution space in polynomial time. The complexity is a linear function of the population size (P), the number of generations (Gen), and the complexity of the fitness function (FF). Other software pipelining algorithms use heuristics to produce a schedule. By making assumptions about the solution space, however, heuristic methods run the risk of getting caught in a local minima. The heuristic methods are polynomial because they reduce the solution space some way. A potentially good area of the solution space, therefore, is ignored by the heuristic. This is especially dangerous if the heuristic is currently sampling a false peak in the solution space. If the discarded area contains the optimal solution, it will never be found. The GA reduces that risk by sampling many points in parallel. When a heuristic cannot perform well, often a GA can. It is not bound



PI \ Iterations	1	2	3	4	5
1	3,6,7				
2	4,8	7			
3	1,5,9	8			
4	2,10	9			
5	11	3,10			
6		4,6,11	7		
7		5	8		
8		1	6,9		
9		2	3,10	7	
10					
11			1,11	8	
12			2,4	6,9	
13			5	3,10	7
14				1,11	8
15				2,4	6,9
16				5	3,10
17					1,11
18					2,4
19					5

PI \ Iterations	1	2	3	4	5
1	3,6,7				
2	1,8	7			
3	2,4,9	8			
4	5,10	3,9			
5	11	4			
6		5,6,10			
7		1,11	7,8		
8		2	6		
9			1,3,9	7	
10			2,10	8	
11			4,11	3,9	
12			5	4,6	
13				1,5	7
14				2,10	8
15				11	3,9
16					4,6
17					1,5
18					2,10
19					11

(a) (b) (c)

Figure 2: Exl: (a) Date Dependency Graph (b) GA Schedule (c) Regular Schedule

by any decision rules. It is stochastic, not deterministic. A GA can, therefore, adapt itself to the problem at hand.

The petri net solution to software pipelining has the property that operations are scheduled as early as data dependencies and resource conflicts allow. In [RA93], it is shown that in the absence of resource conflicts, the petri net produces an optimal schedule. However, the conflicts present in real machines cause an occasional non-optimal schedule to be produced in every heuristic software pipelining technique.

We seek t^ perfuım truth preserving transformations on the dependency graph with a Genetic Alg .ithm. We will let the GA increment the *min* time on an arc. This has the effect of overriding the greedy earliest firing rule by delaying nodes. To evaluate the quality of a point in the search space, we run the petri net algorithm on the modified dependency graph and measure the effective initiation interval. The search space is *sound* as only legal schedules are produced. It is not known whether the space is complete (i.e., all possible legal schedules could be generated).

2 Results and Conclusions

Consider the example of Figure 2(a) in which the minimum initiation interval (without conflicts) is 6/2 = 3. Assume nodes 1, 4 and 10 conflict, for example, because each operation uses a resource of which there is only one copy. Using the genetic algorithm we are able to achieve an initiation interval of 3 (which is optimal) as shown in Figure 2(b). The new loop is in parallel instructions (PI) 11 through 13. With the petri net alone we get a loop body of 4 as shown in Figure 2(c).

Table 1 shows the results of a number of test cases comparing the original petri net, Lam's algorithm, and the Genetic Algorithm. The column entitled EII is the effective initiation interval. The example of Figure 2 is shown as Exl in the table. The schedules

Table 1: Improvement By Genetic Algorithm

	Petri	Lam	GA	Change	
Example	EII	EII	EII	Petri	Lam
Ex1	4.00	4	3	+1.0	+1
Ex2	4.50	6	4	+.5	+2
Ex3	4.33	4	4	+.3	0
Ex4	2.50	2	2	+.5	0
Ex5	4.20	4	4	+.2	0
Ex6	5.67	5	5	+.7	0
Ex7	3.25	3	3	+.3	0
Ex8	3.33	3	3	+.3	0
Ex9	6.50	5	5	+1.5	0
Ex10	6.90	8	6	+.9	+2
Ex11	3.00	2	2	+1.0	0
Average	4.38	4.18	3.73	.65	.45

represent cases in which the genetic algorithm is able to improve the results of the regular petri net schedule. Many of these are examples in which the petri net is worse than Lam, and the GA is able to achieve Lam's results. In some cases, the GA has the best schedule. Other cases not shown include those in which the petri net is better than or equal to Lam, but the GA found no improvement. These results demonstrate the ability of the Genetic Algorithm to find the global best schedule.

Though the petri net algorithm is a powerful software pipelining technique which compares very favorably with other scheduling techniques, we are able to improve the results by applying a genetic algorithm. In the near future, we will optimize the genetic algorithm to this specific problem. We will run the algorithm on a wide variety of problems and analyze the effectiveness of the algorithm under such factors as resource scarcity. We will also examine whether petri net transformations (such as adding a pacemaker [RA93]) will affect the results.

References

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, Inc., Menlo Park, CA, 1989.

[GWN91] G.R. Gao, W-B. Wong, and Q. Ning. A Timed Petri-Net Model for Fine-Grain Loop Scheduling. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation,* pages 204–218, June 26-28 1991.

[Lam88] M.S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* pages 318–328, Atlanta, GA, June 1988.

[RA93] M. Rajagopalan and V. H. Allan. Efficient Scheduling of Fine Grain Parallelism in Loops. *Proceedings of the 26th Annual International Symposium on Microarchitecture,* 24(1):2–11, December 1993.

[RF93] B. R. Rau and J. A. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing,* 7:9–50, 1993.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Parallel Compilation on Associative Computers

Chandra R. Asthagiri[a] and Jerry L. Potter[b]

[a]Computer and Information Science Department, Cleveland State University, Cleveland, OH 44115, USA

[b]Mathematics and Computer Science Department, Kent State University, Kent, OH 45422, USA

Abstract: This paper presents two optimizing associative parallel compiling techniques.

Keyword Codes: C.1.0; C.1.2
Processor Architectures, General; Multiple Data Stream Architectures (Multiprocessors)

1 Introduction

This paper briefly presents two compiling techniques that were developed and implemented using the associative computing (ASC) model[4] to improve compilation speed. Refer the survey article[6] for parallel compilation on various other parallel architectures.

With a massively parallel associative SIMD computer[1] (such as the Active Memory Technology DAP or the Loral Defense System ASPRO) and simple tabular data structures as its basic components, the ASC model supports the concept of processing an entire file of data in parallel. In effect, the records are mapped onto the rows of the tabular data structure, and each row is dedicated with a processor (PE). This model contains a number of constant time associative operations such as associative search, data parallel comparison and numeric computation, and certain built-in functions such as *maxdex/mindex* to find the maximum/minimum of a field, and the *sibdex* to find the minimum value larger and the maximum value smaller than a given input value. These operations take $\log n$ time on the conventional MIMD models. See Potter [4] for details about the architecture and uses of this model including a discussion of constant time operations.

2 Associative Parallel Lexing

The associative lexing decomposes the source program into tokens and stores them in the same field of parallel memory. It assumes that one input character is stored per processor and each source token is surrounded by at least one delimiter (e.g., a blank) on either

[1] An associative SIMD computer emulates associative memory aspects via associative search, but is more powerful than associative memories in the sense that the responding items are processed in situ.

		STEPs 1 and 2										STEP 3					
	<——Ttoken——>												tklen during iteration				
		<——token——>															
		Shift No											0	1	2	3	4
code		1	2	3	4		bi	tkbi	tklen		Token	tkbi					
1		a	b	c	d	?	1	1	?		abcd	1	0	1	2	3	4
2	a	b	c	d	ϵ	?	1	0	?								
3	b	c	d	ϵ		?	1	0	?								
4	c	d	ϵ		e	?	1	0	?								
5	d	ϵ		e	f	?	1	0	?								
6	ϵ		e	f	g	?	1	0	?								
7		e	f	g	h	?	1	1	?		efgh	1	0	1	2	3	4
8	e	f	g	h	i	?	1	0	?								
9	f	g	h	i	ϵ	?	1	0	?								
10	g	h	i	ϵ		?	1	0	?								
11	h	i	ϵ		m	?	1	0	?								
12	i	ϵ		m		?	1	0	?								
13	ϵ		m		n	?	1	0	?								
14		m		n	o	?	1	1	?		m	1	0	1	1	1	1
15	m		n	o		?	1	0	?								
16		n	o		?	?	1	1	?		no	1	0	1	2	2	2
17	n	o		ϵ	?	?	1	0	?								
18	o		ϵ	?	?	?	1	0	?								
19		ϵ	?	?	?	?	1	0	?								
20	ϵ	?	?	?	?	?	1	0	?								
21	?	?	?	?	?	?	0	0	?								

Notation: ϵ = NEWLINE, ? = undefined

Figure 1: Lexing

side. If the maximum length of a token is N (N is 4 in Fig. 1), it does the following:
1. It makes N parallel shifts[2] of 8 bits to the right and 1 word up (It moves one character up laterally so that at the i^{th} shift ($1 \leq i \leq N$) all of the i characters of each word are moved up and aligned in the same row). After the N^{th} shift, each of the tokens of length N or less is in the array word that contains the left delimiter in the initial position. The result is a 2-dimensional table (Ttoken field in Fig. 1) of n rows and $N+1$ columns of source characters, where n is the number of characters in the source input. The first column of the table (marked by 1 in bi field) is the entire set of input characters loaded initially into the parallel memory. The remaining columns contain shifted characters.
2. It recognizes all rows with tokens (Any row in the table, containing a delimiter in the leftmost position and a non-delimiter character in the second position, contains a token.) and marks them as active token entries in parallel (tkbi = 1 on rows 1,7,14, and 16).
3. It initializes the length of all tokens to 0, recognizes the leftmost token in all token rows in parallel and fills the rest of the token rows with zero (null character). As it is scanning all of the token rows in parallel to recognize the end of tokens (a space or N^{th} non-null consecutive character), it increments the length of all tokens by one if their current character is non-null and once the end of token is recognized, the length remains fixed. Now, the array contains only tokens (See the right half of Fig. 1) and these tokens need not be contiguous for later phases of compilation.

The execution of each of the above steps depends on the fixed maximum field width of a token, not on the length of the input. This constant time complexity is an improvement over the $\bigcirc(\log n)$ (n is the number of input characters) time complexity of the thus far

[2] Note: This algorithm assumes a one dimensional grid interconnection network between adjacent cells.

reported best parallel lexing algorithm[3, 6].

3 Associative parallel Expression Compilation

In order to use associative parallelism to its fullest, the expression tokens with attributes such as the structure code[3](SC), associativity (Ac), commutativity (Com), priority (Pr), and precedence (Prec[4]) are stored vertically in a parallel array (See Fig. 2). The associated items of a token are stored in a common associative cell which includes a dedicated PE.

Token at reduction					SC	Pr	nes	Prec	Com	Ac	opr	ID	Proc at reduction					ıo.	Quad.
0	1	2	3	4									0	1	2	3	4		
a	a	a	T_1	T_1	010000	0	0	0	0	0	0	1	0	0	0	0	0		
+	+	+			020000	9	0	9	1	1	1	0	0	0	0	1	1	3	+ a T_1 T_1
b	b	T_1			030000	0	0	0	0	0	0	1	0	0	0	1	1		
−	−	−	−		040000	9	0	9	2	1	1	0	0	0	0	0	1	4	− T_1 c T_1
c	c	c	c		050000	0	0	0	0	0	0	1	0	0	0	0	1		
±	±				060000	9	0	9	1/2	1	1	0	0	0	1	1	1	2	± b T_1 T_1
d	T_1				070100	0	1	0	0	0	0	1	0	0	1	1	1		
+					070200	9	1	22	1	1	1	0	0	1	1	1	1	1	+ d e T_1
e					070300	0	1	0	0	0	0	1	0	1	1	1	1		

Note: Five columns under Token/Proc denote a single field of Token/Proc at different iterations
AC=1: Left associative, AC=2: Right associative, Com=1:commutative, Com=2:Weakly non-commutative

Figure 2: Reduction of Expression '$a + b - c \pm (d + e)$' in Associative Cells

The associative compilation uses massive parallel associative search in place of conventional shift-reduce procedure to generate optimized intermediate code for an expression in a single pass. It selects the cell with one of the highest precedence operators[5] by searching all cells for attribute type 'operator' (opr) and then searching for the maximum precedence. Both searches are performed in parallel in one step each. Then it selects the operands for the operator in one parallel search using the sibdex function. Then it reduces the selected triple by generating an intermediate quadruple, 'opr operand1 operand2 result' with result being a temporary variable to hold the value of the operation ('+ d e T_1' in Fig. 2), replacing operand1 of the source code by result ('$a + b - c \pm T_1$', column 1 of Token field in Fig. 2), and inactivating both operand2 and operator (Proc field entry becomes 1 in Fig. 2). Then it continues the evaluation on both sides of the recent result temporary as long as the evaluation of the new operator does not violate the mathematical axioms on priority and the same result temporary can be used for all of the results in the sequence, thereby minimizing the number of temporaries needed. For example, it treats '$a + b - c \pm T_1$' logically as '$a + b \pm T_1 - c$' and reduces it into '$a + T_1 - c$' (column 2 of Token field in Fig. 2). Notice that both sequential and Fischer's non-canonical parallel parsing [2] methods need two temporaries to reduce the same expression (Fig. 3).

[3] The structure code is a unique integer representation showing the structural information (such as position and nesting level) of each source token (See Potter[4].). The use of structure codes allows all of the tokens in the program to be searched in parallel.

[4] Precedence (Prec) for all operators in an entire expression/program is computed in parallel using the equation 'Prec[$] = Pr[$] + nes[$] * k', where k is a constant denoting the priority for parenthesis '$' denotes the entire parallel field. See [1] for details on precedence computation.

[5] If the highest precedence operator is left associative, pick the leftmost occurrence of it, otherwise pick the rightmost occurrence of it.

Sequential Method			*Fischer Method*		
Expression	No	quadruple	Expression	No	quadruple
$a+b-c \pm (d+e)$	1	+ a b T_1	$a+b-c \pm (d+e)$	1	+ d e T_1
$T_1 - c \pm (d+e)$	2	- T_1 c T_1	$a+b-c \pm T_1$	2	+ a b T_2
$T_1 \pm (d+e)$	3	+ d e T_2	$T_2 - c \pm T_1$	3	− T_2 c T_2
$T_1 \pm T_2$	4	± T_1 T_2 T_1	$T_2 \pm T_1$	4	± T_2 T_1 T_2
T_2			T_2		

Figure 3: No Register Optimization

This single pass associative compilation of an expression into an optimized intermediate code with linear time complexity [1] is better than all other multi pass parallel approaches with $\bigcirc(\log n)$ (n is the number of input tokens) time complexity (for parsing) and linear time complexity (for optimization). This $\bigcirc(\log n)$ time complexity excludes the steps in restructuring expressions to a form suitable for parallel evaluation on MIMD machines and the steps in the overhead of synchronization and communication between processors. Also, disk I/O being often the limiting factor, the single pass feature of the associative algorithm is a significant feature. Thus, the linear time complexity of this one pass associative algorithm using no recursion or pointers is better than the linear complexity of the three pass (intermediate code generation, code labeling, and code reordering in intermediate code representation[5]) sequential algorithms using extensive data structures.

4 Conclusion

The algorithms discussed in this paper, parallel lexing and optimal expression compilation, use associative techniques to achieve constant time operation. The authors feel that many other areas of compilation can also benefit from associative techniques.

References

[1] C. Asthagiri, *An Associative Parallel Compiler for an Associative Computing language,* Ph.D Dissertation, Kent State University, Aug. 1991.

[2] C. N. Fischer, "On Parsing and Compiling Arithmetic Expressions on Vector Computers," *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 2, pp. 203-224, 1980.

[3] W. D. Hillis, and G. L. Jr. Steele, "Data Parallel Algorithms," *Comm. ACM*, Vol. 29, No. 12, pp. 1170-1183, 1986.

[4] J. L. Potter, *Associative Computing - A Programming Paradigm for Massively Parallel Computers,* Plenum Press, 1992.

[5] R. Sethi, and D. Ullman, "The generation of Optimal code for Arithmetic expressions," *J. ACM*, Vol. 17, No. 4, pp. 715-728, 1970.

[6] D. B. Skillicorn, and D. T. Barnard, "Compiling in Parallel," *Journal of Parallel and Distributed Computing*, Vol. 17, pp. 337-352, 1993.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Partitioning of Variables for Multiple-Register-File Architectures via Hypergraph Coloring*

A. Capitanio[a] and N. Dutt[b] and A. Nicolau[b]

[a]Dipartimento di Elettronica ed Informatica, Universita' di Padova, 35121 Padova, Italy

[b]Information and Computer Science Department, UC Irvine, 92717-3425, CA, USA

Abstract: Multiple-functional-unit architectures allow to boost performances by executing many operations concurrently but register file technology can provide only with a limited I/O data-bandwidth, able to support a few, fast, pipelined functional units.

One of the possible solutions to increase the available bandwidth is the adoption of multiple register banks. In this paper we present a technique for partitioning variables onto such a non-homogeneous register space. The technique is based on a hypergraph model and allow a partial rescheduling of the code when a legal partitioning cannot be found.

Keyword Codes: C.1.1, D.m
Keywords: Single Data Streams Architectures. Software, Miscellaneous.

1 Introduction

In a multiple-functional-unit architecture performance is the result of two key parameters: the number of operations simultaneously executed and the execution cycle-time. These parameters are correlated since the cycle-time depends, also, on the data-path delays through the Register File (RF); RF timings are influenced by the number of registers in it contained and its number of ports - 2 read-ports and 1 write-port per Functional Unit (FU) are required in order to sustain peak performance. Hence the need for a design tradeoff among the number of registers, the number of functional units (FU) and the cycle-time.

The adoption of multiple register banks can mitigate this bottleneck by decreasing the I/O bandwidth requirement for each RF; henceforth allowing to create architectures with a larger number of FUs. These processors require variables partitioned among the register banks but standard register allocation techniques [2] are not suited to perform this kind of allocation because of the different *locality* of distribution of resources (See [1]).

Register allocation techniques frequently rely on the concept of Virtual Registers (VR): initially operands are mapped onto an ideal register file containing an infinite number of VRs, this is later constrained to the number of available registers. In case of multiple

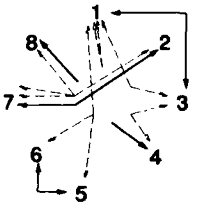
*This work was partially supported by Esprit Project 9072-GEPPCOM, ONR N0001486K0215 and the UCI Faculty Research Grant

VLIW Code

LD R5 R1 1000	LD R6 R1 2000
R7 <- R5 * R6	R2 <- R1 + 4000
R8 <- R8 + R7	ST R7 R2 100
R1 <- R1 + 8	R3 <- R3 + 1
R4 <- 1000	---
Rcc <- R3 > R4	---

LD Rx Ry k = Load memory loc Ry+k into Rx
ST Rx Ry k = Store Rx into memory loc Ry + k

Interference Hypergraph



HG = {HN,HE}

HN = {1,2,3,4,5,6,7,8}

HE = {{1,1}read,{1,3}r,
{2,8,7,7}r,{3,4}r,
{5,6,1}r,
{1,3}write,{4}w,
{8}w,{6,5}w,{7,2}w}

COLORING
RED = {1,2,4,5,8}
BLUE = {3,6,7}

Figure 1: A well-colored Hypergraph

RFs we assume this operation to be performed by mean of a two-step process: 1) each operand is assigned to a VR which is allocated to a RF, 2) each RF is constrained to its actual size.

Our technique can be used to solve the first phase, hence we do not make any assumption about the RFs' size, in that, a later phase will adequately take care of the allocation within each bank. In our model the process of assignment of VRs to RFs is solely constrained by the limited number of ports available per RF.

2 Partitioning of Variables

We assume, as a model of execution, a horizontally microcoded, multi-functional-unit architecture - a Very Long Instruction World architecture [3]. The model is parameterized in the number of register files (R) and the number of functional units (F); and it provides 2F read ports and F write ports evenly distributed among the RFs (i.e., each RF has $\frac{2F}{R}$ read-ports and $\frac{F}{R}$ write-ports).

Given a program compiled for a single-RF VLIW we must partition its variables into R sets, each to be allocated on a RF; this can be done by means of a hypergraph HG = (HN,HE). A hypergraph HG is defined as a set of hypernodes, HN, and a set of hyperedges, HE; each hyperedge being defined as a subset of the set of hypernodes. Each hypernode in HN is associated to one and only one VR in the code, and each hyperedge in HE models the competition among VRs concurrently accessed within an instruction.

Figure 1 presents a VLIW code fragment and its Interference Hypergraph. VRs $\{R1..R8\}$ are associated to hypernodes $\{1..8\}$ and read(write) hyperedges are represented as *stars* of dashed(solid) arrows, each one connecting the set of hypernodes associated to VRs accessed in one instruction. In this model colors are used to represent the RF in which a VR is contained, hence the number of nodes of the same color contained in a read(write) hyperedge is equivalent to the number of operands concurrently read(written) from(in) a same RF.

A read-hyperedge containing no more than $\frac{2F}{R}$ nodes of the same color is said to be *well-colored*, and is representative of an instruction which access no more than $\frac{2F}{R}$ operands stored in the same RF and that, therefore, can be legally executed[2].

[2] Write operations are dealt with in the same fashion, but for the different number of write-ports available: $\frac{F}{R}$.

```
procedure COLOR-HYPERGRAPH;
for each hEdge ∈ HE $\boxed{1}$
  LABEL: DeferredOP
  for each node ∈ nodesOf(hEdge) $\boxed{2}$
    if (NOT isColored(node)) $\boxed{3}$
      min = MAXINT;
      minColor = UNCOLORED;
      for each color ∈ COL
        number = maxNumberColored(node,color);
        if (number < min)
          minColor = color;
          min = number;
        end if
      end for
      if (minColor == UNCOLORED)
        op = chooseAnOpToDefer(hEdge,node); $\boxed{4}$
        deferOp(op); $\boxed{5}$
        GOTO DeferredOP;
      else
        colorNode(node,minColor); $\boxed{6}$
      end if
    end if
  end for
end for
```

Figure 2: Hypergraph Coloring Algorithm

In Figure 1 the Interference Hypergraph of code produced for a 2-wide VLIW is well-colored with 2 colors - the code can be legally executed on a 2-RFs VLIW.

2.1 The Coloring Algorithm

The algorithm (See Fig. 2) is based on three nested loops. The outermost one sequentially picks one hyperedge (hEdge) at a time ($\boxed{1}$) and for each hypernode contained in hEdge ($\boxed{2}$), and not already colored ($\boxed{3}$), it assigns a color ($\boxed{6}$) that minimizes the maximum number of nodes with the same color in any hyperedge of the hypergraph.

When the algorithm runs into a situation in which a hypernode is not colorable ($\boxed{4}$), since any color, legal within hHedge, would make other hyperedges illegal, then the code must be locally rescheduled to permit the continuation of the process. This is achieved by deferring an operation ($\boxed{5}$), chosen among those that access the VR associated to one of the uncolorable hypernodes. If the process is not possible, for lack of resources or because of existing dependencies, then new (empty) instructions are introduced, and the code is locally recompacted.

The deferral of the selected operation may require other operations to be moved, since the presence of anti/output-dependencies may prevent the transformation. When such a situation is detected the transformation tries to defer first the operation at the bottom of the chain of dependencies and if a cycle of dependencies is found this is broken by

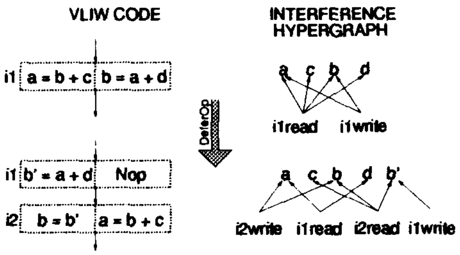


Figure 3: Deferring of an operation within a circular dependence

inserting a copy operation. (see Figure 3).

A useful optimization of the process is obtained by ordering the set of hyperedges according to the probability of execution of the associated instructions; the process of coloring becomes progressively more constrained hence by following this order we increase the probability of having to resort to a code transformation in less frequently executed sections of code.

The coloring algorithm was applied to a set of 13 benchmarks, representative of common scientific code, previously compiled for a single-register-file VLIW architecture; we took into considerations 3 configurations: a 8-wide VLIW with 4 and 2 RFs, and a 4-wide VLIW with 2 RFs (See [1]).

In most of the cases it was possible to assign each VR to a RF without the need for altering the code, and even when the algorithm had to transform the code the relative performance degradation was always below the 5% of simulated run-time.

3 Conclusions

A possible solution to mitigate the RF requirements in terms of I/O bandwidth in a VLIW architecture is the adoption of multiple register banks; however producing code for this kind of microprocessors requires operands partitioned onto a non-homogeneous register space.

In this paper we have proposed a model for partitioning variables onto multiple RFs via hypergraph coloring; we have also supplied an algorithm for this task which is able to partially reschedule code when a legal solution cannot be found. Experimental evidence suggest that code produced by the proposed technique presents little performance degradation when compared with code produced for a single-RF VLIW.

References

[1] A. Capitanio, N. Dutt and A. Nicolau. Toward Register Allocation for Multiple Register File VLIW Architectures. Technical report, ICS dept., UC, Irvine, TR 94-6, 1994.

[2] G.J. Chaitin. Register Allocation and Spilling via Graph Coloring. In *Proceedings of the SIGPLAN 1982, PLDI*, pages 98–105, Jun 1982.

[3] J.A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, (No. 7):pages 478–490, 1981.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)
1994 IFIP.

Realizing Parallel Reduction Operations in Sisal 1.2

Scott M. Denton, John T. Feo and Patrick J. Miller

Computer Research Group, Lawrence Livermore National Laboratory [1], Livermore CA

Abstract: Often the tasks of a parallel job compute sets of values that are then reduced to a single value or gathered to build an aggregate structure. In this paper, we present compilation techniques that recognize pairs of computation–reduction expressions in Sisal 1.2. We present performance results that demonstrate the utility of our techniques.

Keyword Codes: D.1.1; D.1.3; D.3.3
Keywords: Applicative (Functional) Programming; Concurrent Programming; Language Constructs and Features

1 Computation–reduction expressions

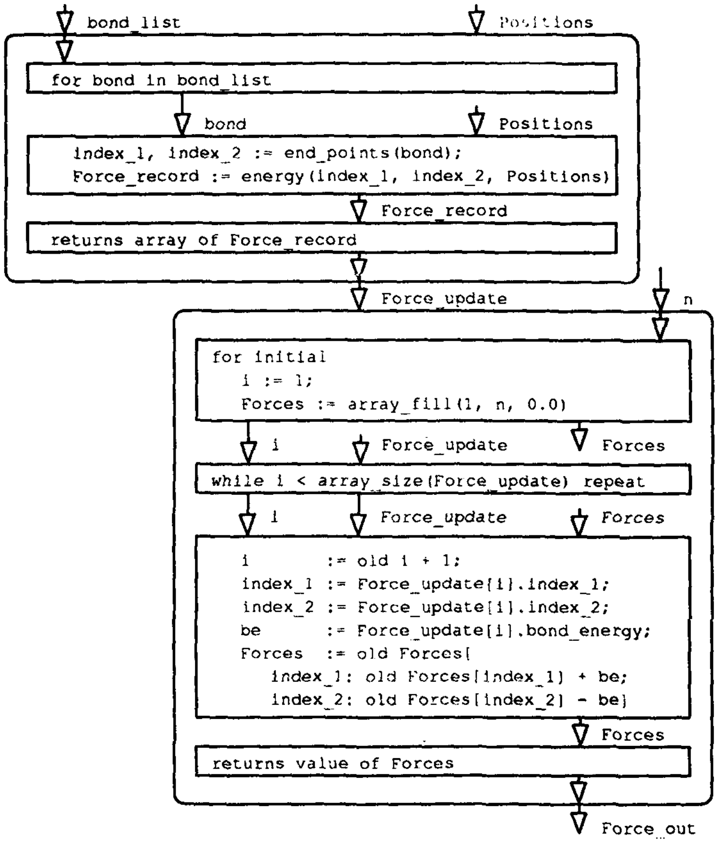
Pairs of computation–reduction expressions appear frequently in parallel programs. First, the computation expression computes a set of values, and then the reduction expression reduces the set of values to a single value or gathers the values to build an aggregate structure. An array sum reduction returns a scalar value by adding together all array elements. A histogram is a transformational reduction that counts the number of occurrences of a value in one array and stores the count in another. A recurring theme in particle physics codes is the calculation of bond forces. The forces between particles are calculated, and then the force incident on each particle is computed. Since pairs of computation–reduction operations appear frequently in application programs, they are good targets for optimization. The efficient expression and implementation of these pairs of operations can reduce the cost of parallel programming.

Since reduction expressions may introduce dependencies, most languages separate the computation and reduction tasks. Typically, the programmer must write two loops. In Sisal 1.2 [1], the user must write a *for* expression to compute the values and a *for initial* expression to reduce the values. In this paper we present optimization techniques to implement pairs of computation–reduction expressions in Sisal 1.2 as single parallel loops.

[1] This work was supported by Lawrence Livermore National Laboratory under DOE contract W-7405-Eng-48. This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California. and shall not be used for advertising or product endorsement purposes.

This optimization overlaps computation and reduction, reduces runtime overhead, and reduces storage requirements.

A common pair of computation–reduction expression occurs when computing the forces between a set of particles. Consider a set of *n* particles and *m* bonds in a *bond_list.* Each bond represents a force between two particles. We want to calculate the force of each bond and then accumulate the force incident on each particle. The following figure depicts the Sisal code and the IF1 [2] graph structure of the computation and reduction expressions.



```
for bond in bond_list
```

```
index_1, index_2 := end_points(bond);
Force_record := energy(index_1, index_2, Positions)
```

```
returns array of Force_record
```

```
for initial
   i := 1;
   Forces := array_fill(1, n, 0.0)
```

```
while i < array_size(Force_update) repeat
```

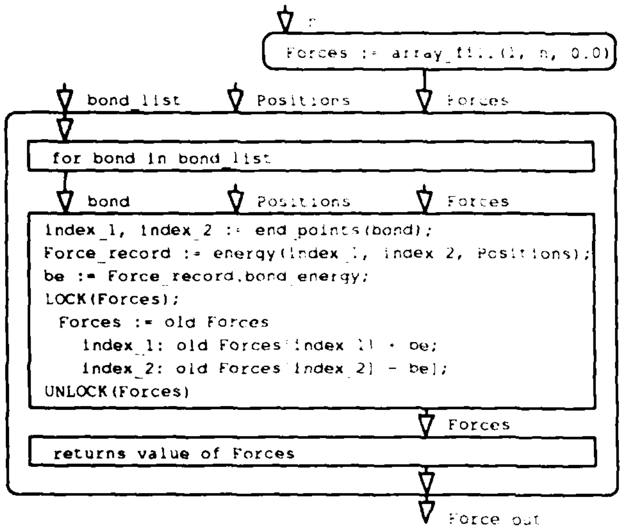
```
i        := old i + 1;
index_1  := Force_update[i].index_1;
index_2  := Force_update[i].index_2;
be       := Force_update[i].bond_energy;
Forces   := old Forces[
   index_1: old Forces[index_1] + be;
   index_2: old Forces[index_2] - be]
```

```
returns value of Forces
```

The top node, the computation expression, is a parallel *for* expression. It has three subgraphs: generator, body, and returns. The generator defines a set of index values. An instance of the body is executed for each value, and it computes a *Force_record.* The functions *end_points* and *force* return the indices of the two particles participating in the bond and the force of the bond, respectively. The force records are passed to the gather subgraph that gathers them into the array *Force_update.*

The bottom node, the reduction expression, is a sequential *for initial* expression. It has four subgraphs: initial, test, body, and returns. The initial subgraph initializes the index value and the *Forces* array. The body is executed once for each force record, and updates the force on two particles. The returns subgraph selects the final value of *Forces* and passes it out from the compound node as the array *Force_out*. We refer to this implementation as unoptimized.

We have extended the Sisal compiler to merge the operations of the bottom node into the top node, thereby, eliminating the overhead of the *for initial* expression and the array of *Force_records*. The compiler generates the following pseudo-code and IF1 graph structure,



```
Forces := array_fill(1, n, 0.0)

for bond in bond_list

index_1, index_2 := end_points(bond);
Force_record := energy(index_1, index_2, Positions);
be := Force_record.bond_energy;
LOCK(Forces);
  Forces := old Forces
     index_1: old Forces[index_1] + be;
     index_2: old Forces[index_2] - be];
UNLOCK(Forces)

returns value of Forces
```

The first node initializes *Forces* and passes it to the second node, a parallel *for* expression. Its generator is identical to the generator of the original *for* expression, and its body and returns subgraphs are compositions of the body and returns subgraphs, respectively, of the original *for* and *for initial* expressions. Since *Forces* is a shared resource, the compiler places a lock in the generated code about any read and write accesses to insure mutual exclusion. We refer to this implementation as optimized.

Currently, the Sisal compiler merges only those pairs of *for* and *for initial* expressions that meet the following five criteria:

1. The *for initial* expression depends directly on the *for* expression, and does not depend on any descendent of the *for*.

2. The initialization clause of the *for initial* expression is independent of the array of values consumed.

3. The *for initial* expression depends on the *for* expression for only an array of values.

4. The *for initial* expression consumes every value of the array of values, once and in order.

5. The *for initial* expression has no loop carried dependencies other than an index value and the shared accumulator.

The shared accumulator refers to the scalar value or aggregate structure returned by the reduction expression. If the first criterion is not met then the *for initial* expression could not execute until the descendent computation completed preventing us from merging the *for* and *for initial* expressions. The second criterion permits us to move the initialization clause before the *for* expression. Criteria three, four, and five guarantee that the i-th iteration of the *for initial* expression depends only on the i-th iteration of the *for* expression, permitting us to merge the bodies of the two expressions. Since the compiler does not test for commutativity, the optimized implementation may be nondeterminate.

2 Conclusions

We ran a series of experiments to evaluate the performance of our optimizations. We used computation–reduction expressions from molecular dynamics similar to the expressions used in the previous section. As expected, the optimized code with locks in the parallel loop used less space than the unoptimized code. It also executed on average five to twenty percent faster; however, we did not always see appreciable performance gains.

There are many factors that influence the execution time of the optimized code versus the execution time of the unoptimized code. If the size of the computation expression is at least p (number of processors) times greater than the size of the reduction expression, then there is little lock contention. Essentially, the concurrent tasks contend for the lock the first time and then become staggered, arriving at the critical section at different times. However, the time to execute the locks can become significant for large problems. We wrote a version in Sisal 1.2 that built a *Force* array per worker and then merged the individual arrays at the end of the computation. This reduced the number of locks from one per bond to one per worker resulting in a significant space and time saving.

size	100000	200000	300000
sequential reduction	12 MB 9 sec	23 MB 19 sec	35 MB 28 sec
parallel reduction	8 MB 8 sec	17 MB 16 sec	25 MB 24 sec

processors	1	2	3	4
lock within parallel loop	2.9 MB 48 sec	2.9 MB 28 sec	2.9 MB 20 sec	2.9 MB 17 sec
temporary with merge	3.1 MB 35 sec	3.6 MB 19 sec	3.9 MB 14 sec	4.3 MB 11 sec

References

[1] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. *SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2.* Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[2] S. Skedzielewski and J. Glauert. *IF1: An Intermediate Form for Applicative Languages.* Manual M-170, Lawrence Livermore National Laboratory, Livermore, CA, July 1985.

[3] S. Denton, J. Feo, and P. Miller. *Parallel Implementation of Reduction Operations in Sisal 1.2.* In progress, Lawrence Livermore National Laboratory, Livermore, CA, April 1994.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)
1994 IFIP.

An Introduction to Simplex Scheduling

Benoît Dupont de Dinechin

C.E.A., Centre d'Études de Limeil-Valenton, 94195 Villeneuve St Georges cedex France

Abstract: Simplex scheduling refers to an approach where the central scheduling problems originating from instruction scheduling and modulo scheduling are solved on a simplex tableau (a matrix) instead of a scheduling graph. By definition, a central scheduling problem is a subproblem of the original scheduling problem, where all the resource constraints have been removed, and where some extra precedence constraints have been included to materialize the fact that some of the tasks have already been scheduled.

By using a parametric version of the simplex algorithm, we optimally solve central scheduling problems involving symbolic valuations of the scheduling graph edges, such as linear expressions of a yet unknown software pipeline initiation interval. In addition, by using a lexicographic cost function and by introducing extra equations in the simplex tableau, we also compute optimum solutions to the central scheduling problems in polynomial time, while simultaneously minimizing the cumulative register lifetimes.

Keywords: Deterministic scheduling; Instruction scheduling; Software pipelining; Modulo scheduling; Lifetime-sensitive scheduling; Parametric linear programming.

1.1 Why Simplex Scheduling

As a matter of example, let us suppose that we have to build an instruction scheduler based on the list scheduling heuristic. The list algorithm would run on the scheduling graph to build a priority list, usually the latest optimal start times of the tasks (instructions). Then instruction scheduling would take place, according to the rules:

- copy from the priority list the sublist of the ready tasks, i.e. the tasks whose predecessors in the scheduling graph have completed their execution;
- upward scan this sublist, and start at the current time the tasks which do not trigger resource conflicts with any of the previously scheduled tasks;
- if the sublist is empty, or if no further scheduling can take place at the current time without triggering resource conflicts, increment the current time;
- remove the scheduled tasks from the priority list, and iterate the whole process until the priority list is empty.

While such a simple process is guaranteed to succeed if the scheduling graphs has no cycles, it must be significantly extended to cope with the cycles which appear in the context of software pipelining of recurrent and `while` loops. In particular, every time a task of a cyclic graph is scheduled, the margins[1] of every not yet scheduled task must be

[1] Recall that the set of allowed start times of a task in a central problem is an interval, whose lower and upper bounds are respectively called the *left margin* and the *right margin* (of the task).

recomputed. Otherwise there is no way to detect that the problem has turned infeasible, a situation which arises whenever there are no start times within the margins of a task which prevent it from triggering resource conflicts with the already scheduled tasks.

Another drawback of list schedulers is that they exhibit a greedy behavior which has not received a practical solution beyond scheduling the instructions backwards. Generally speaking, the more subgoals one assigns to a list scheduler beyond minimizing schedule length, the more one has to refine the heuristics which computes the sublist of the ready tasks, often with unreliable results. At some point, backtracking must be introduced [2], a solution we want to avoid even if it were to achieve register lifetime sensitive scheduling.

1.2 Principles of Simplex Scheduling

Let T be the yet unknown initiation interval of a software pipeline we try to build by modulo scheduling. Let $\mathcal{V} = \{\sigma_i\}_{1 \leq i \leq N}$ be the set of the nodes of the scheduling graph, that is, the instructions of the loop body. Let $\mathcal{E} = \{(\sigma_i, \sigma_j, b_k - \beta_k T)\}_{1 \leq k \leq M}$ be the set of the edges of the scheduling graph. The positive coefficients β_k, which are denoted Ω in the papers by Rau & al. , are non-zero only in the cases of loop-carried dependencies.

By definition of a scheduling graph, each arc $(\sigma_i, \sigma_j, b_k - \beta_k T)$ materializes a constraint $t_j - t_i \geq b_k - \beta_k T$. One may therefore express all the precedence constraints as:

$$\begin{bmatrix} u_1^1 & \cdots & u_1^N \\ \vdots & & \vdots \\ u_M^1 & \cdots & u_M^N \end{bmatrix} \begin{bmatrix} t_1 \\ \vdots \\ t_N \end{bmatrix} \leq \begin{bmatrix} -b_1 & \beta_1 \\ \vdots & \vdots \\ -b_M & \beta_M \end{bmatrix} \begin{bmatrix} 1 \\ T \end{bmatrix} : (\sigma_i, \sigma_j, b_k - \beta_k T) \in \mathcal{V} \Longrightarrow \begin{cases} u_k^i = +1 \\ u_k^j = -1 \\ u_k^l = 0_{l \neq i, l \neq j} \end{cases}$$

There is nothing especially clever in the above formulation, beyond the fact that we can now use parametric linear programming to solve the problem of minimizing t_N, without having to guess T. Indeed computing the minimum value of T for which the problem is feasible is a natural by-product of the parametric simplex algorithm (see § 1.4). This is a polynomial-time linear program because $U = [u_i^j]$ is the sign-inverted transpose of the incidence[2] matrix of the scheduling graph, so this matrix is totally unimodular[3] [3].

However, such a matrix formulation becomes really interesting when one realizes that extra equations can be introduced to minimize the cumulative register lifetimes. To understand how, let us assume that the arcs $(\sigma_i, \sigma_{j_{i_1}}, b_{k_{i_1}} - \beta_{k_{i_1}})$, $(\sigma_i, \sigma_{j_{i_2}}, b_{k_{i_2}} - \beta_{k_{i_2}})$ ... carry a register value defined by σ_i to $\sigma_{j_{i_1}}$, $\sigma_{j_{i_2}}$ .... Let l_i be a new variable which represents the lifetime of the register value written by σ_i. Then, by adding the equations $t_{j_{i_1}} + \beta_{k_{i_1}} T - t_i \leq l_i$, $t_{j_{i_2}} + \beta_{k_{i_2}} T - t_i \leq l_i$ ... to the constraints, we can make cumulative register lifetime minimization a secondary goal beyond minimizing t_N. This requires the lexicographic minimization of the bi-dimensional cost function $[t_N, \sum l_i]^T$.

Minimizing lifetimes is useful, at least because it prevents the solver from scheduling the instructions too early, and is practical too. Indeed the augmented problem can still be solved in polynomial time, since the augmented constraint matrix U' is also totally unimodular. The demonstration of this nice property comes from the observation that U' can be obtained from U by duplicating colums, then rows, and by clearing elements of the resulting matrix in a way which allows the application of a result by Truemper [4].

[2] The incidence matrix $A = [a_i^j]$ of a graph has one column for each arc, and one row for each node; in this matrix a_i^j is -1 if the arc j leaves the node i, $+1$ if the arc j enters the node i, and 0 otherwise.

[3] By definition a matrix with integral coefficients is totally unimodular if all its subdeterminants are equal to $+1$, -1, or 0. Incidence matrices are guaranteed to be totally unimodular.

1.3 A Lexico-Parametric Dual Simplex Algorithm

A *lexico-parametric linear program* is a problem defined by:

$$\text{lexico-minimize } Zu = C^T x, x \in P(u) = \{x | x \geq \vec{0}, Ax \leq Bu\}$$

where $x \in \mathbf{Q}^n$, $A = [a_i^j] \in \mathbf{Q}^{m \times n}$, $B = [b_i^j] \in \mathbf{Q}^{m \times p}$, $u \in \mathbf{Q}^p$, $C = [c_i^j] \in \mathbf{Q}^{n \times q}$, and $Z = [z_i^j] \in \mathbf{Q}^{q \times p}$. In this formulation, $x = (x_1, \ldots, x_n)^T$ is the vector of unknowns, $u = (u_1 = 1, u_2, \ldots, u_p)^T$ is the vector of *parameters* whose first component is always 1, A is the *constraint matrix*, B is the *constant matrix*, C is the *cost matrix*, and Zu the *economic function value*. An *integer lexico-parametric linear program* has the same formulation, but the components of A, B, C, u, x, Z are further restricted to integer values.

Under the traditional formulation of linear programming, $p = q = 1$, that is, B and C are vectors, and Z is a scalar. Parametric linear programming problems were introduced by Feautrier who formulated and implemented the general method so solve them, both over the rationals and the integers [1]. The lexicographic extension of linear programming problems, whether parametric or not, is simple matter. The only extra work implied is the need to lexicographically compare the columns of C^T, instead of comparing scalars.

To solve lexico-parametric linear programs, we shall use an extended version of the so-called *dual simplex* algorithm [3]. At any point of the resolution process, all the data relevant to the simplex algorithm is maintained in the so-called *simplex tableau*:

$$\begin{array}{ccccc}
[\quad \tilde{Z} \quad] & = & [\quad \tilde{C}^T \quad] & & [\quad \tilde{J}^T \quad] \\
\begin{bmatrix} \\ \tilde{B} \\ \\ \end{bmatrix} & = & \begin{bmatrix} \\ \tilde{A} \\ \\ \end{bmatrix} & & \begin{bmatrix} \\ \tilde{I} \\ \\ \end{bmatrix} \\
(1 \ \ldots \ u_{p-1} \ u_p) & & (y_1 \ y_2 \ \cdots \ y_n) & & (y_{n+1} \ \cdots \ y_{n+m})
\end{array}$$

Initially, $\tilde{A}$ is A, $\tilde{B}$ is B, $\tilde{C}$ is C, $\tilde{I}$ is the unit matrix, $\tilde{J}$ and $\tilde{Z}$ are zero matrices. The vector y is the concatenation of $(x_1 \ldots x_n)$, and of the *slack variables* $(s_1 \ldots s_m)$. The slack variables are used to convert the inequalities of the linear program into equalities.

The lexico-parametric dual simplex algorithm maintains the invariant that the columns of the matrix $\tilde{C}^T$ are always lexico-positive, while the linear forms $\tilde{B}_i u$ are initially of arbitrary sign. The objective of the dual simplex algorithm is to make these linear forms positive or zero by pivoting, or to detect that the linear program is infeasible. In general computing the signs of $\tilde{B}_i u$ is the tough part of parametric linear programming. However in our application to instruction scheduling the components of the parameter vector u always have a known value at any time, so the *context* [1] is reduced to simple equalities.

To select a pivot a_p^q, we first select among the constant matrix $\tilde{B}$ a strictly negative line $\tilde{B}_p u$. Then, among the reduced cost matrix $\tilde{C}^T$, a column q is selected such that:

$$\frac{\tilde{C}_q}{a_p^q} = \text{lexicographic} \max_{a_p^j < 0} \left\{ \frac{\tilde{C}_j}{a_p^j} \right\}$$

If there is no negative $\tilde{B}_p u$, the problem is solved and the current solution is optimum. If there is no $a_p^j < 0$ for any negative $\tilde{B}_p u$, the problem is infeasible.

1.4 Application to Instruction Scheduling

In our application of lexico-parametric linear programming to instruction scheduling, $(x_1 \ldots x_n)$ is $(t_1 \ldots t_N, l_1 \ldots l_{n-N})$, $A = U'$, B is such that $B_i = (-b_i, 0, \beta_i)$, and $u =$

$(1, S, T)$. The parameter T is the initiation interval, while the parameter S is used to probe the set of the allowed start times of the task currently selected for scheduling.

Before starting to schedule instructions, an *initial central schedule* is built in order to compute the minimum value of T, called $T_{recurrence}$, which makes the central problem feasible. Remember that a dual simplex algorithm finds the problem to be infeasible whenever there exists a $\tilde{B}_p u < 0$ such that $a_p^j \geq 0 \ \forall j$ in the constraint matrix. In this case the only way to make the problem feasible is to make $\tilde{B}_p u = \tilde{\beta}_p T - \tilde{b}_p$ positive. If $\tilde{\beta}_p > 0$, we assign to T the value $(\tilde{b}_p - 1)/\tilde{\beta}_p + 1$, and resume the search for a variable p to leave the basis, otherwise the problem is infeasible. The final value of T is $T_{recurrence}$.

Once a suitable value of $T \geq T_{recurrence}$ is available, the yet unscheduled tasks are *selected* one after the other by a higher-level driver in order to be scheduled. The fact that a task σ_i is selected to be the current one to schedule is represented by adding the pair of inequalities $t_i \leq -S \wedge t_i \leq S \Leftrightarrow t_i = S$ to the central problem. The set of the allowed start times for the current task σ_i can now be explored by assigning a value to S in the simplex tableau, and then by running the lexico-parametric dual simplex algorithm.

In practice, after zero or a few pivoting steps, either the problem is infeasible, meaning that S is outside the current margins of σ_i, or the new optimal central schedule under the constraint $t_i = S$ is obtained. In the latter case, assuming that the value the outer driver assigned to S did not make σ_i conflict on a resource basis with the previously scheduled tasks, then scheduling σ_i at $t_i^* = S$ is admissible. This is translated in the simplex tableau by *freezing* S, that is, converting any linear form $\lambda_k^1 + \lambda_k^2 S + \lambda_k^3 T$ in the left hand side of the simplex tableau (i.e., $\tilde{B}$ and $\tilde{Z}$) to $(\lambda_k^1 + \lambda_k^2 t_i^*) + \lambda_k^3 T$.

A nice feature of simplex scheduling is the valuable information it maintains for free. For instance, whenever the central scheduling problem is infeasible, there exists a $\tilde{B}_i u < 0$ such that $a_i^j \geq 0 \ \forall j$. Since the constraint matrix $\tilde{A}$ is totally unimodular at any stage of the simplex algorithm [3], its coefficients are always -1, 0, or $+1$. Consequently the non-null a_i^j in the line i are equal to 1. Remember also that each inequality in a linear program is associated to a slack variable $s_l, 1 \leq l \leq m$ in the simplex tableau.

Therefore, by listing the j such that y_j is a slack variable, and that $a_i^j = 1$, one can find which are the inequalities whose sum yields to the condition impossible to satisfy. The way we build the linear program implies that these inequalities must come from the precedence constraints, and not from the lifetime equations. It turns out that the arcs associated with these inequalities are precisely the arcs of the current critical cycle.

References

[1] P. Feautrier: "Parametric Integer Programming": *RAIRO Recherche Opérationnelle / Operations Research*, Vol. 22, Sept. 1988.

[2] R. A. Huff: "Lifetime-Sensitive Modulo Scheduling": *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, Albuquerque, June 1993.

[3] A. Schriver: "Theory of Linear and Integer Programming": Wiley, 1986.

[4] K. Truemper: "Unimodular Matrices of Flow Problems with Additional Constraints": *Networks*, Vol. 7, 1977.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Speculative Evaluation for Parallel Graph Reduction

James S. Mattson Jr.[a] and William G. Griswold[b]

[a]Department of Computing Science, Glasgow University, 17 Lilybank Gardens, Glasgow G12 9SZ, Scotland

[b]Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Drive, La Jolla, California 92037-0114, USA

Abstract: Speculative evaluation can improve the performance of parallel graph reduction systems through increased parallelism. Although speculation is costly, much of the burden can be absorbed by processors which would otherwise be idle. Despite the overhead required for speculative task management, our prototype implementation achieves 70% efficiency for speculative graph reduction, with little impact on mandatory tasks. Through speculative evaluation, some simple benchmarks exhibit nearly a factor of five speedup over their conservative counterparts.

Keyword Codes: D.1.1; D.1.3.
Keywords: Programming Techniques, Applicative (Functional) Programming; Concurrent Programming.

1 Motivation

Graph reduction is a popular implementation technique for non-strict functional programming languages. Because graph reduction is free of side-effects, it is well-suited to parallel processing. With conservative evaluation, an expression is not evaluated until its results are actually required. Speculative evaluation increases parallelism by assigning potentially useful computations to idle processors before their results are required. If a speculative computation later becomes *necessary*, the time required to complete the computation is reduced. On the other hand, if a speculative computation later becomes *irrelevant*, the time wasted on its evaluation is unimportant, because the only processors that engage in speculative computations are those that would otherwise have been idle.

Unfortunately, speculative tasks are difficult to manage [2]. Speculative tasks compete with mandatory tasks for processors. A speculative task which becomes necessary should be upgraded to a mandatory task. A speculative task which becomes irrelevant should be discarded. Shared subgraphs may have to be repaired whenever a speculative task is terminated prematurely.

Speculative evaluation may contribute additional hidden costs as well. For instance, even when processing elements are available, the memory and communications demands of speculative tasks may degrade the performance of mandatory tasks through increased contention.

The key to an effective speculative evaluation strategy is to shift most of the burden of speculation to the processors that would otherwise be idle. By reducing the impact of speculation on mandatory tasks, speculative evaluation can be used to improve the performance of some programs without significantly degrading the performance of others.

2 Concepts

A simple scheduling policy is inadequate for speculative graph reduction. Speculative tasks may perform unnecessary computations, and the program may terminate with a number of speculative tasks outstanding. Clearly, speculative tasks should only be considered for execution when no mandatory tasks are available [1]. Therefore, a speculative graph reduction system must provide prioritized, preemptive task scheduling to ensure that mandatory tasks are not deferred while some processors are engaged in speculative computations. Furthermore, some speculative tasks are more likely to become useful than others. Any reasonable scheduling strategy for speculative tasks must favor the tasks that are most likely to become useful. Therefore, a range of speculative priorities are necessary, with the highest priorities assigned to the most promising tasks.

Because of the constraints on task granularity, a single task is typically responsible for the reduction of several nodes in the program graph. Each node may be shared among a different set of tasks, so the priority of a speculative task may require adjustment each time it enters or leaves a node. However, an enormous amount of bookkeeping is involved in a priority system that keeps detailed accounts of sharing [5].

To avoid the overhead of such detailed bookkeeping, we adopt a lazy approach to priority adjustment. Whenever a task enters an updatable node, it locks the node, so that other tasks that try to enter the node will block until the result is available. If a task blocks on a node locked by a lower priority task, the priority of the running task is raised to that of the blocking task. Whenever a task completes the evaluation of a node on which equal priority tasks are blocked, it recalculates its priority as the maximum of its initial priority and the priorities of any tasks that are blocked on other nodes it still has locked.

With this approach, sharing is not considered relevant until another task actually demands the shared result. Consequently, very little overhead is required for dynamic priority adjustments. Although this approach does not ensure the optimal assignment of priorities to speculative tasks, it does guarantee that a task will never be blocked on a lower priority task.

Some form of speculative task throttling is necessary to avoid swamping the system with speculative tasks. However, the throttling mechanism must ensure that crucial parallelism will not be lost when a speculative computation becomes necessary. In particular, should a speculative task *spawn* a child task of equal priority, the scheduler must upgrade the child if its parent becomes necessary.

When a node is evaluated conservatively, the result is written back into the program graph, overwriting—and destroying—the graph for the original expression. With speculation, however, the standard update mechanism can result in unbounded growth of the program graph. Unless the speculative results become completely unreachable, they cannot be reclaimed by standard garbage collection techniques. Partridge suggests that a possible solution to this problem is to revert evaluated parts of the program graph to their unevaluated form if they are not yet necessary and if they occupy less space as unevaluated expressions [5].

Update reversion requires a special speculative update mechanism that does not destroy the original graph. When a speculative task completes a computation, it overwrites the original node with a *deferred update* node that contains pointers to both the unevaluated expression and the result. When a speculative task enters a deferred update node, it just follows the pointer to the result. However, when a mandatory task enters a deferred update node, it performs the update and then follows the pointer to the result. If memory is exhausted, the garbage collector can free additional space by reverting some of the deferred updates to their unevaluated form.

3 Implementation

A prototype implementation containing these and other ideas has been developed for the BBN TC2000 "butterfly". It is based on the Spineless Tagless G-Machine model of graph reduction [4]. Few modifications to the STG-Machine are necessary to support speculative evaluation [3].

Conservative task synchronization is handled with the aid of special "black hole" nodes. When a task enters a updatable node, it locks the node by replacing it with a black hole. Any task that enters a black hole is suspended and added to the black hole's blocking queue. When a black hole is updated, any tasks that were blocked on the black hole are resumed.

If a speculative task enters an updatable node, the system must be prepared to restore the original node should the speculative task be terminated prematurely. Therefore, when a speculative task enters an updatable node, it replaces it with a revertible black hole, or a "grey hole.". A grey hole contains not only a pointer back to the original expression, but also a task identifier and the speculative task's stack depth when it entered the node. The latter two items are used to temporarily increase the scheduling priority of the speculative task should a task with a higher scheduling priority block on the grey hole. When a grey hole is updated, any blocked tasks are resumed, and the updating task downgrades its own priority if appropriate.

Each speculative task has two priorities: a base priority and a working priority. The base priority is assigned when the speculative task is created, and only changes if the root of the speculative task is demanded by a higher priority task. The working priority is used for scheduling, and may change as other tasks block on nodes that are locked by the speculative task. Whenever a mandatory task blocks on a node that a speculative task has locked, the speculative task is upgraded to mandatory. If the shared node is also the root node for the speculative task, the base priority of the speculative task is changed to mandatory as well. However, if the shared node is nested deeper within the speculative computation, the task is only temporarily upgraded until it completes the evaluation of the shared node. If multiple mandatory tasks block on different nodes that are locked by the same speculative task, the speculative task is temporarily upgraded until the topmost node is evaluated.

The stack depths that are recorded in the grey holes are used for quick identification of the topmost node at which a mandatory task is blocked. An upgraded speculative task records the stack depth of the topmost grey hole on which a mandatory task is blocked, to reduce the overhead required for determining when to downgrade the task back to a speculative priority.

Whenever a speculative task blocks on a lower priority speculative task, it raises the priority of the other task to match its own. A speculative task may block several other speculative tasks of varying priorities and at varying depths in its computation, so the

Table 1: Speculative and Prescient Speedups (wrt Conservative)

	2 PEs		4 PEs		8 PEs		16 PEs		32 PEs	
Benchmark	Sp	Pr	Sp	Pr	Sp	Pr	Sp	Pr	Sp	Pr
pqueens	0.94	1.1	0.98	0.97	0.98	1.0	1.1	1.3	1.1	1.2
primes	1.5	1.9	2.6	3.6	3.7	6.8	4.7	11.5	3.9	15.9
sieve	0.95	0.95	0.70	0.69	0.73	0.71	0.77	0.74	0.87	0.85
tautology	1.7	1.8	2.7	2.8	4.2	3.9	4.3	3.8	3.4	3.0

shallowest-stack-depth solution used for mandatory upgrades is inadequate for handling nested priority changes. To guarantee that the priority of a speculative task is always the maximum of its own base priority and the priorities of any blocked speculative tasks, a speculative task must be able to recognize when it has awakened the highest priority blocked task and adjust its own priority accordingly. Therefore, each time a speculative task updates a grey hole and awakens a task of the same priority, it walks up its own stack to determine the highest priority of any speculative tasks that are still blocked on one of its grey holes. Although this operation is expensive, only speculative tasks incur the additional overhead, so conservative reduction does not suffer.

We have tested the prototype implementation on a small set of benchmarks, and the initial results are promising. For each of the benchmarks, we devised a corresponding "prescient" program that uses conservative parallelism to evaluate only those tasks that are necessary for program completion and to evaluate all necessary tasks as soon as possible. The prescient program achieves idealized speculative evaluation: perfect prediction of the program's future needs and early execution of necessary tasks with minimal overhead.

References

[1] F. Warren Burton. "Speculative computation, parallelism, and functional programming," *IEEE Transactions on Computers*, C-34(12):1190-1193, December 1985.

[2] Paul Hudak. "Distributed task and memory management," *Second Annual ACM Symposium on Principles of Distributed Computing*, pp. 277-289, Montreal, Quebec, Canada, 17-19 August 1983, ACM Press.

[3] James S. Mattson Jr. *An Effective Speculative Evaluation Technique for Parallel Supercombinator Graph Reduction*, PhD thesis, Department of Computer Science and Engineering, University of California, San Diego, February 1993.

[4] Simon L. Peyton Jones. "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine," *Journal of Functional Programming*, 2(2):127-202, April 1992.

[5] Andrew S. Partridge. *Speculative Evaluation in Parallel Implementations of Lazy Functional Languages*, PhD thesis, Department of Computer Science, University of Tasmania, October 1991.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Toward a General-Purpose Multi-Stream System[1]

Avi Mendelson[a] and Bilha Mendelson[b]

[a]Dept. of Electrical Engineering, Technion - Israel Institute of Technology, Haifa 32000, Israel. mendlson@ee.technion.ac.il

[b]IBM Israel - Science and Technology, MATAM - Advanced Technology Center, Haifa 31905, Israel. bilha@vnet.ibm.com

Abstract: This paper presents a new approach for a general purpose multi-stream system. The system is designed to support parallel execution of different processes/threads. We achieve this goal by using a unique hardware architecture that supports the use of our new "semi-dynamic" stream scheduling. The architectural principles presented here can be applied to both VLIW and supper-scalar systems. The system can support synchronization primitives, dynamic interleaves of execution streams and other operating system operations. Simulations indicate that the system can utilize a large number of execution streams with relatively small overhead.

KEYWORDS: VLIW, Super-scalar, Multi-stream, Thread

1 Introduction

Modern computer architectures can integrate several functional units in the same system. Future computer architectures will be able to integrate a large number of such units. However, The utilization of such systems is limited by the capability of the programmer to write "massively parallel" programs (coarse grain parallelism) and by the capability of the compiler/hardware to exploit the *instruction level parallelism (ILP)* that can be found in each instruction stream. The multi-stream approach advocates using both techniques in order to take full advantage of such systems. The XIMD [3] machine offers to extend the VLIW approach. Different threads and their related ILP are scheduled at compile time (assuming some run time support), thus, streams cannot dynamically interleaves at run time. A different approach [1] calls to extend the super-scalar approach by forming a global scoreboard that can manage different streams concurrently. Streams here can dynamically be interleaved, but the implementation may require very complicated hardware.

The capability for dynamic scheduling of execution streams is very important for supporting a multi-threaded and multi-processing environment. Recently, different researchers report that the ILP degree of programs changes from one program to another, and in many cases the degree of parallelism changes from one execution phase to another [2].

[1] Supported in part by Ollendorff foundation.

This paper presents a "semi-dynamic" execution scheduling. Each stream is broken into Execution Blocks (EB), and the system can support parallel execution of different execution streams. The system allocates a fixed set of resources for each EB for its entire execution, but can dynamically change the allocation at the beginning of each EB.

In order to support the semi-dynamic scheduling approach, a new hardware component, termed *coordinator* is introduced. The coordinator aims at allocating the resources to the different execution streams, but may also be used for implementing synchronization primitives and other operating system activities.

2 Compile time support

We assume that the programmer uses threads to indicate the streams that can be executed in parallel. The compiler divides each thread (or process) into EBs (presently, this process is performed manually but we are looking for heuristics for optimal partitioning). Each EB ends with a call to another EB (*e.g.*, if it needs a system call), swaps (replaces its execution with another EB) or waits for synchronization. The call directive supports parameter passing as well as serves as an indication of the return EB. Synchronization is handled by the system coordinator which manipulates synchronization conditions and activate streams as needed.

Fine-grain parallelism is provided by static scheduling of each EB according to the amount of resources and/or the maximal parallelism that can be extracted for it. Global scheduling techniques such as software pipelining, trace scheduling or global instruction scheduling can be applied to each EB with no restrictions.

The compiler attaches a header to each EB that indicates the resources it needs. The header contains the type and quantity of the resources. The process uses a *process header* which indicates the EBs that have to be invoked when the process is initiated.

3 The Hardware Model

Figure 1 shows the general structure of the proposed architecture. Two main features distinguishes our new approach from other multi-stream architectures: (1) the sequencer in our model controls only the resources which were attached to the current EB, and (2) a new hardware component, termed a *system coordinator*, is introduced. The system coordinator interleaves the execution of the different EBs, attaches the required resources to an EB for its entire activity, and supports different synchronization primitives. The execution of each EB is independent. Thus, a delay in executing one EB will not have any direct effect on the performance of others.

The system presented in Figure 1 is designed to support M concurrent execution EBs. The hardware is composed of N functional units (FU) combined into an *FU-file*, M sequencers, coordinator units and a register file.

Each sequencer can access all the resources attached to it. All the sequencers can be operated in parallel since the FU-file is multi-ported and each functional unit can access any register. Conflicts of access to registers are prevented by allocating a different set of registers to each sequencer. Note that only the coordinator can allocate resources.

The sequencer in our system is similar to the one used in the "traditional" VLIW or the score-board in a super-scalar architecture. To support the dynamic nature of our system, logical resource numbers are used. At compile time, the compiler determines how many resources should be allocated to each stream. It schedules the instructions

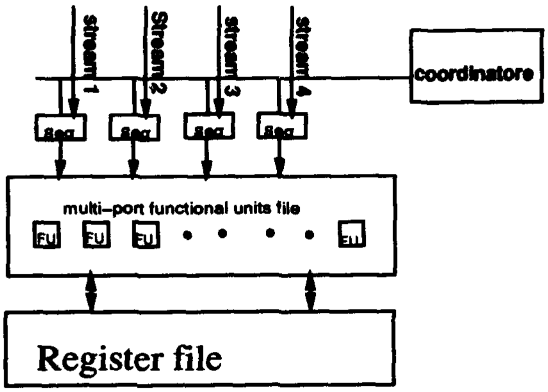


Figure 1: The proposed multi-stream architecture

according to this assignment and uses logical numbers for the allocated resources. At run time, the coordinator allocates the resources needed, and provides the physical numbers to the sequencer before the execution of the EB starts. The sequencer holds a set of mapping registers for accessing the right resources.

The system coordinator is also used for implementing different synchronization primitives and other "basic operating system operations" which are vital for correct operation of multi-threaded environments.

4 Run time support

During run time, the system coordinator controls the activities of the entire system. The execution of an application starts by sending the process header to the system coordinator. The system coordinator puts the EBs listed in the process header into the *ready stream pool.* It finds EBs that are in the ready stream pool which can best utilize the machine and sends them to the sequencers for execution. The coordinator guarantees that the EB will not be interrupted during its execution.

When an EB is completed, it releases all of its resources and the coordinator schedule another EB for execution. The current EB is inserted into the ready stream pool or into the waiting queue (if synchronization is required). Note that the system coordinator cannot schedule an EB for execution unless it can allocate the resources it needs. The type and quantity of resources the EB requests are used for that purpose. When it finds a proper candidate from the ready stream pool, it removes the EB from the pool, allocates the resources for the EB, and sends their physical identification to the sequencer. The sequencer maps the logical resource identifications used by the compiler to the physical resources.

5 Supporting Operating System Operations

The system we are presenting can efficiently support a multi-process environment where each process has its own address space. The process can be divided into threads which are operated in parallel and share the same address space.

The system supports basic synchronization primitives such as semaphores, barriers and communication among the different EBs. When an EB terminates its operation it provides an indication to the coordinator why it was terminated. If the EB indicates that it needs a synchronization event or any other system service the coordinator will not re-schedule it before this condition was fulfilled. This way we can ask the coordinator to perform some of the "traditional" OS operations such as I/O, or the coordinator can be used to communicate with some other host that will perform these operations.

6 Summary and Conclusions

Current VLSI technology can offer a larger number of functional units. Unfortunately, the software cannot efficiently utilize them. To overcome this disadvantage, this paper outlined the basic characteristics of a new multi-stream machine. The system is capable of supporting fine-grain parallelism found in the streams, and coarse grain parallelism as directed by the use of threads. The threads are broken into EBs, and each EB may need different number of resources for its execution. The system dynamically exchanges execution of EBs so that it can achieve better machine utilization. The use of a multi-stream architecture with dynamic streams exchange reduces the effect of execution stall on the stream being executed. It allows the integration of asynchronous system operations, such as handling I/O devices together with execution of CPU intensive applications.

Currently we are looking on different tradeoffs in designing such a system. We are considering reducing the complexity of the register file and the functional units by limiting the number of resources each sequencer is connected to. We are looking at the effect of different scheduling mechanisms on the utilization of the system and optimal ways to break the execution stream of the thread into EBs.

References

[1] S.W Keckler and W.J Dall. "Processor coupling: Integrating compiler time and runtime scheduling for parallelism," *The 19th Annual Int. Symp. on Computer Architecture*, pp. 202–213, May 1992.

[2] D. W. Wall. "Limits on instruction-level parallelism," In *ASPLOS IV: 4th Inter. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 176–188, April 1991.

[3] A. Wolfe and J. P. Shen. "A variable instruction stream extension to the vliw architecture," *ASPLOS IV: 4th Inter. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 2–14, April 1991.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Representing Control Flow Behaviour of Programs

Cyril Meurillon and Ciaran O'Donnell[1]

Departement Informatique, *Telecom Paris*, 46 *rue* Barrault, 75634 Paris Cedex 13, France

1 Introduction

Many program constructs are considered difficult to optimize because they are random. For example, the irregular behaviour of an `if` statement in the middle of a loop hinders optimization.

The underlying idea of this paper is that there is nothing random about program behaviour. It is true that randomness can often be observed, for example in a profile. However, by focusing attention on what is essential in a measurement a hidden meaning can be made to emerge from the apparent randomness.

Optimization by compilers is often based on the abstract representation of the program. Using an information theoretic characterization of behaviour, we derive laws about programs that cannot be inferred from the representation alone. Optimization in hardware has concentrated on *ad hoc* improvement of behaviour. A scientific characterization improves understanding of why a technique performs better than another.

Figure 1 shows a loop in a program and the corresponding control flow graph (CFG). Nodes such as D are special because the value (`true/false`) of their *condition* can affect subsequent control flow.

```
i=0;
do {
   f(i);
   i++;
} while (i<4);
```

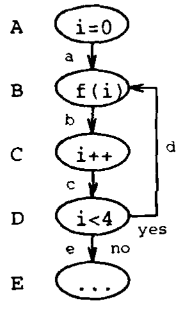


Figure 1: An example of loop and its CFG

Control flow is just the sequence of nodes in the order of their execution. A program trace is the control flow of the nodes that are executed. Note that only conditional nodes are significant in the trace. Control flow can also be represented by the sequence of transitions. James Larus has shown that this is a better representation of control

[1] Franco Gasperoni participated in this. It came out of a talk in his office and we pursue it together.

flow and that the *dominator tree* can be used to minimize the number of transitions recorded [Lar93].

Larus makes a side remark concerning a technique credited to Christopher Fraser in which control flow is represented simply by a string of bits. Transitions at a condition node are represented by a single bit: 1 if the condition is true, 0 otherwise. The CFG is used to determine the node following a condition, depending on the previous history of the trace.

The second idea, which we call the "Compress Hypothesis", is that the *entropy* (information content) of *Fraser's trace* can be used as an indicator of the branch predictability (regularity or randomness) of a program. Note that use of Fraser's trace is essential as it is a minimal representation of control flow. A node trace, for example, contains nodes other than conditional nodes that are redundant. The encoding of transitions in Larus's trace also introduces redundancy. In these two cases, the redundancy due to the encoding affects entropy and interferes with redundancy due to program behaviour.

In order to validate the "Compress Hypothesis", we decided to first use a data compression program such as COMPRESS or GZIP on Fraser traces of the SPEC benchmarks and compare the compression ratios that result. These programs certainly do not give the best indicators of predictability possible but they provide a good starting point and are easy to put in practice. In order to collect the branch history of a program, we wrote a very simple preprocessor, which is invoked to source compilation. It takes a C source as input and modifies it so that a Fraser trace is saved in a file at run-time. No special compiler or run-time environment is necessary.

The idea is to find the control flow statements (`if`, `for`, `while`, `do while`) in the source, and incorporate function calls so that the result of the condition is recorded.

2 Different kinds of trace

Various behaviour representations of the C loop of figure 1 — the node trace, transition trace and Fraser's trace — are shown in Figure 2. The node and the transition traces contain redundant information: node C always follows node B in the node trace, and transition c always follows transition b in the transition trace. A Fraser trace eliminates redundancy as it focuses on changes in the control flow.

nodes	A B C D B C D B C D B C D E
transitions	a b c d b c d b c d b c e
Fraser	1 1 1 0

Figure 2: Different kinds of trace

Larus's dominator technique for eliminating redundancies from a transition trace still contains redundancies due to the node number encoding.

A Fraser trace can be conveniently represented as a vector b, b_i designating the ith bit in the trace. The portion of Fraser trace of length n and ending at time T can then be expressed by the sequence $b_{T-n+1}b_{T-n+2}\dots b_T$.

3 Entropy

The "Compress Hypothesis" depends on the notion of information content (*entropy*). Shannon first introduced the notion of entropy in [Sha48]. He considers a source as producing n different symbols $S_{i\in\{1..n\}}$ with respective probablities of occurence $p_{i\in\{1..n\}}$. He then defines entropy as :

$$H = -\sum_{i=1}^{n} p_i \ln p_i$$

The higher the entropy, the more the randomness.

Huffman uses Shannon's occurence probability model in his compression algorithm. It is optimum for memoryless sources. This means that the occurence probabilities are neither intercorrelated or selfcorrelated, i.e.

$$p_{ij} = p_i p_j$$

where p_{ij} is the probability that the source produces the sequence $S_i S_j$. Real sources seldom act this way. For example in an English text the three letter sequence "T H E" is more probable than any other.

Many other compression algorithms have been described. LZW is one of the most well known. This algorithm can efficiently deal with combination of symbols, or *words*. The program GZIP implements another algorithm, LZ77, which usually performs better than LZW. There is no simple mathematical expression of the notion of entropy defined by an algorithm such as LZW. The algorithm itself is the simplest definition and the size of the output defines the information content of the input.

4 Signatures

We call a *signature* of length n at time T of an execution, the substring $b_{T-n+1}b_{T-n+2}\ldots b_T$ of a Fraser trace. Signatures correspond to the contents of a window of length n sliding through the trace.

Signatures are a program-oriented indicator of predictability. The number of different signatures in an execution gives information about program randomness: the higher the number, the less predictable the program.

Signatures on their own are not the best indicator of predictability. Consider for example the four bit signature 0001, corresponding to an execution of a four-branch loop (three forward branches in the loop not taken, a backward branch executed to the top of the loop). The repeated execution of this pattern gives rise to four different signatures (0001, 0010, 0100, 1000), each corresponding to different phases of the loop. It is possible to *normalize* the signature by rotating until appearance of the smallest configuration possible (0001).

A second flaw is that the signature is weighted identically, regardless of how often it occurs. For example the loop pattern, 0001, receives the same weight as a pattern that occurs only once. It is a good idea to encode the signatures using Huffman's algorithm. This ponderates signature probabilities and limits noise.

5 Results

In these experiments the number of different signatures of length n encountered in a range $b_0b_1\ldots b_{L-1}$ was measured. Results for $n = 32$, and $L = 1,000,000$ are shown in the first table. The column "random" gives the number of signatures of a randomly generated binary string. Note how different the program figures are compared to the random case. It is clear that a signature is an indicator of control flow behaviour.

random	espresso (a)	li	eqntott
999,966	52,576	9,991	6,793

signatures for $n = 32$ and $L = 1,000,000$

L	random	espresso (a)	li	eqntott
100,000	99,968	4,514	7,109	2,655
1,000,000	999,966	52,676	9,991	6,793
10,000,000	?	135,680	17,391	14,060

signatures for L varying ($n = 32$)

n	espresso (a)	li	eqntott
32	52,576	9,991	6,793
48	116,114	16,856	32,295

signatures for n varying ($L = 1,000,000$)

O	espresso (a)	li	eqntott
0	64,337	14,801	≈ 11,000
1,000,000	52,576	9,991	6,793

signatures for O varying ($n = 32$ and $L = 1,000,000$)

	espresso (a)	li	eqntott
non-norm.	52,576	9,991	6,793
norm.	23,523	?	2421

effect of normalisation

	espresso (a)	li	eqntott
signatures	135,680	17,391	14,060
code length	7.78	10.74	9.73

Huffman's code length

Table 1: Results

We next change the signature length, n, and the sample size, L. Results are shown in the next two tables. Again, signatures are seen to be a good characterization of program behaviour. Their number only grows by 3, not by 10, when L is multiplied by 10 (as opposed to the random case). Similarly, if the signature size is increased by 16 bits, the number is only multiplied by 3.

It is now possible to make these measurements clearer. Firstly, we have eliminated "startup noise.". The method chosen was to eliminate the first 1,000,000 branches ($O = 1,000,000$) in the experiments reported. The fourth table shows that the number of signatures with noise ($O = 0$) is significantly higher.

The fifth table shows that normalization allows to divide the number of signatures by a factor of 2-3. Unnormalized ponderated signatures using Huffman's algorithm are shown in the final table. It is very significant that the signatures of ESPRESSO are encoded with less bits (7.78) than those of LI (10.74), even though ESPRESSO had 8 times more signatures. ESPRESSO is hence more predictable.

References

[Lar93] James R. Larus. Efficient program tracing. *IEEE Computer*, 26(5), May 1993.

[Sha48] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, July-October 1948.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Transformations on Doubly Nested Loops

Ron Sass and Matt Mutka

Department of Computer Science, A-714 Wells Hall, Michigan State University, East Lansing, MI 48824-1027, {sass,mutka}@cps.msu.edu.

Abstract: A new algorithm, the *phase method*, is described in this report. The phase method is a loop transformation based on unimodular transformation theory. The phase method is not limited to perfectly nested loops.

Keyword Codes: D.3.0; D.1.3; D.3.m
Keywords: loop transformations, restructuring compilers, unimodular transformations

1 Introduction and Background

The main result presented in this paper is a new algorithm, the phase method. It is closely allied to other unimodular transformation algorithms such as Banerjee's [2] except for the important distinction that it accepts imperfectly nested loops. This result is significant for two reasons. Although other techniques exist to convert imperfect loop nests into perfect loop nests — loop distribution being the most notable — they cannot always be applied or it may be undesirable to apply them. Secondly, some researchers are proposing alternative frameworks because unimodular transformations are limited to perfectly nested loops[3].

We begin by letting S, P, and E represent any sequence of non-loop statements. Figure 1(a) represents perfectly nested loops and Figure 1(b) shows the class of loops we parallelize. We call P the leading statements and E the trailing statements. Banerjee defines index point, iteration space, data dependences, and dependence graph in [2]. We use those concepts here except we expand the iteration space to include leading and trailing statements.

We use the algorithm for fine-grain parallelism Banerjee presents in [2]. The algorithm produces new loop bounds — m_1, M_1 and $m_2(), M_2()$ — and a unimodular transformation matrix, U. We frequently refer to the skew (u_{12} component of U) as μ.

2 Algorithm

The most important feature of the phase method is that it determines when to execute the leading and trailing statements at compile time. If unmodified, the dependence analyzer does not consider our expanded iteration space. We can extract the data dependences by directly augmenting the input to the constraint solver within the dependence analyzer, as Wolf does in [5]. The two additional constraints are: P executes when $i_2 = n_2(i_1)$ and E executes when $i_2 = N_2(i_1)$. This means that all dependence vectors have length two. Thus, all dependences can be represented in the form shown in Figure 1(c).

```
DO  $i_1 = n_1, N_1$
   DO  $i_2 = n_2, N_2$
      $S[i_1, i_2]$
   END
END
```

(a)

```
DO  $i_1 = n_1, N_1$
   $P[i_1]$
   DO  $i_2 = n_2, N_2$
      $S[i_1, i_2]$
   END
   $E[i_1]$
END
```

(b)

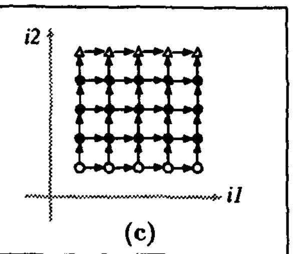
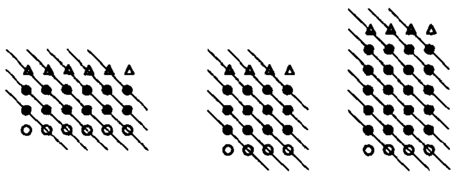


Figure 1: perfect and imperfect loop nests; expanded iteration space.

Next, we invoke the fine-grain parallelism algorithm. Although the algorithm was meant for perfectly nested loops, the dependence information that we are passing includes the leading and trailing statements' dependences. The remaining task is to generate code, but first the algorithm takes into account that the index points in the iteration space are not necessarily the same sequence of statements. The phase method does this by recognizing that the "wave" (the outer loop of the transformed code) travels through the iteration space in phases. For a doubly-nested loop there are three phases. The first phase may be vacuous or both leading and body statements are executed. The middle phase has three exclusive possibilities: only body statements are executed; leading, trailing, and body statements are executed; or it is vacuous. The third phase is similar to the first. The same wave on the three different iteration spaces shown below, for each case the middle phase is different.



Since the angle of the wave depends on the amount of skewing, it may be the case that in a given phase there may be more waves than leading or trailing statements. That is, the previous examples do not illustrate that the leading statements may only be executed once every n waves during the first two phases.

We identify the phases for a particular iteration space with the variable s. When s is negative, it indicates that the leading, trailing, and body statements are part of the middle phase. When s is positive, the middle phase has only body statements. The number of waves in the middle phase is $|s|$. To determine s, we find the last wave to execute a leading statement (A) and the total number of waves (w_{tot}). After we find s, we determine the number of waves in the first and third phases:

$\text{A} = \Omega = \lvert u_{12} \rvert (N_1 - n_1)$	waves that may execute P or E
$w_{\text{tot}} = \lvert u_{11} \rvert (N_1 - n_1) + \lvert u_{12} \rvert (N_2 - n_2) + 1$	total number of waves
$s = w_{\text{tot}} - 2 \lvert u_{12} \rvert (N_1 - n_1)$	characterizes the middle phase
$\alpha = \omega = \min\{\text{A}, \text{A} + s\}$	number of waves in the first/last phases

```
$P[n_1, n_2 - 1]$
DO $W = m_1, m_1 + \alpha_2 - 1$
  PAR DO $X = m_2(W), M_2(W)$
    $S^U[W, X]$
  END
  IF MOD(W+1,$\mu$)=$\mu$-1 THEN
    $P^U[W + 1, m_2(W + 1) - 1]$
  END
END DO
```

(a)

```
DO $W = m_1 + \alpha_2, m_1 + \alpha_2 + |s| - 1$
  IF MOD($M_1 - W, \mu$)=$\mu - 1$ THEN
    $E^U[W, M_2(W) + 1]$
  END
  PAR DO $X = m_2(W), M_2(W)$
    $S^U[W, X]$
  END
  IF MOD(W+1,$\mu$)=$\mu$-1 THEN
    $P^U[W + 1, m_2(W + 1) - 1]$
  END
END DO
```

(b)

```
DO $W = m_1 + \alpha_2, m_1 + \alpha_2 + |s| - 1$
  PAR DO $X = m_2(W), M_2(W)$
    $S^U[W, X]$
  END
END DO
```

(c)

```
DO $W = m_1 + \alpha_2 + |s|, M_1$
  IF MOD($M_1 - W, \mu$)=$\mu - 1$ THEN
    $E^U[W, M_2(W) + 1]$
  END
  PAR DO $X = m_2(W), M_2(W)$
    $S^U[W, X]$
  END
END DO
$E^U[N_1, N_2 + 1]$
```

(d)

Figure 2: three phases are selected from these templates.

Code generation is simply a matter of generating code for each of the three phases. Several small improvements can be made when generating the phases; we show the simplest one here. For more details, see [4]. The first phase is shown in Figure 2(a). When generating code for the middle phase, s determines the code. If s is zero, then nothing is generated. If $s < 0$ then the code of Figure 2(b) is generated. If $s > 0$ then the code of Figure 2(c) is generated. The last phase is shown in Figure 2(d).

3 Example

Below, an example is listed with line numbers and dependence information:

```
          code
 5 | DO I = 3,10
 6 |   C(I) = 1/(A(I-3,1)*A(I-3,1)-1)
 7 |   COLSUM(I) = 0
 8 |   DO J = 1,10
 9 |     A(I,J) = C(I)*X(I,J)
10 |     COLSUM(I) = COLSUM(I)+A(I,J)
 8 |   END DO
 5 | END DO
```

Dependence Information		
flow	6:C(I) → 9:C(I)	(0,1)
flow	7:COLSUM(I) → 9:COLSUM(I)	(0,1)
outp	10:COLSUM(I) → 7:COLSUM(I)	(0,1)
flow	9:A(I,J) → 6:A(I-3,J)	(3,0)
flow	10:COLSUM(I) → 10:COLSUM(I)	(0,1)

Passing the vector set $\{(0,1),(3,0)\}$ and the loop bounds to Banerjee's algorithm[2] results in the following transformation matrix, $U = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, and new loop bounds, $m_1..M_1 = 4..20$ and $m_2(x)..M_2(x) = \lceil \max\{3, x - 10\} \rceil .. \lfloor \min\{10, x - 1\} \rfloor$. Next we calculate the number of waves in each phase (s, α, and ω) based on the equations given earlier.

$$s = 17 - 2(1)(10 - 3) = 3 \qquad \alpha_2 = \min\{7, 10\} = 7 \qquad \omega_2 = \min\{7, 10\} = 7$$

Using the templates from Figure 2(a,c,d) and the improvements from [4], we get the resulting code.

```
C(3) = 1/(A(0,1)*A(0,1)-1)                    Enter phase 1 ...
COLSUM(3)=0
DO W=4,10
  PAR DO X=MAX(3,W-10),MIN(10,W-1)
    A(X,W-X)=C(X)*X(X,W-X)
    COLSUM(X)=COLSUM(X)+A(X,W-X)
  END DO
  C(MAX(3,W-9)-1)=1/(A( (MAX(3,W-9)-1) -3,1)*A( (MAX(3,W-9)-1) -3,1)-1)
END DO
DO W=11,20                                    Phase 2 and Phase 3 merged...
  PAR DO X=MAX(3,W-10),MIN(10,W-1)
    A(X,W-X)=C(X)*X(X,W-X)
    COLSUM(X)=COLSUM(X)+A(X,W-X)
  END DO
END DO
```

4 Comparisons and Conclusion

Little has been written about imperfect loop nests. Loop distribution is usually successful but it fails when a strongly connected component of the dependence graph is still imperfect. In [1], Abu-Sufah describes a simple technique that is easy to apply and almost always legal, but performs poorly during execution. M.J. Wolfe provides complex conditions to permute imperfect loop nests in [6]. M.E. Wolf [5] discusses unimodular transformations on imperfect loop nests but suggests that his method is only practical for loop skewing. A more detailed comparison is in [4].

In this paper we argued that unimodular transformations are not fundamentally limited to perfect loop nests. We extended the notation of an iteration space to include imperfectly nested loops and presented a unimodular transformation algorithm that generates parallel code for imperfectly nested loops of depth two.

References

[1] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers.* PhD thesis, University of Illinois at Urbana-Champaign, November 1978.

[2] Utpal Banerjee. Unimodular transformations of double loops. In Alexandru Nicolau, David Gelernter, Thomas Gross, and David Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 192–219. Pitman Publishing, 1991.

[3] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-2995.1, University of Maryland, College Park, April 1993.

[4] Ron Sass and Matt Mutka. Transformations on doubly nested loops. Technical Report CPS-93-3, Department of Computer Science, Michigan State University, January 1993.

[5] Michael E. Wolf. *Improving Locality and Parallelism in Nested Loops.* PhD thesis, Stanford University, August 1992.

[6] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman Publishing, 128 Long Acre, London WC2E 9AN, 1989.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

A Comparative Study of Data-Flow Architectures

David F. Snelling[a] and Gregory K. Egan[b]

[a]Centre for Novel Computing, Deptartment of Computer Science, University of Manchester, snelling@cs.man.ac.uk

[b]Laboratory for Concurrent Computing Systems, Swinburne University of Technology, gke@swin.edu.au

Abstract: This paper compares three widely differing Data-Flow systems using a relatively uniform metric which is representative of the actual amount "work" performed. The three systems compared are the Manchester Data-Flow Machine, the Stateless Data-Flow Architecture, and the CSIRAC II machine.

Keyword Codes: C.1.3, C.4
Keywords: Processor Architecture, Data-Flow; Performance of Systems

1 Introduction

In Data-Flow systems, data values rather than being stored at particular addresses, are *tagged*. The tag includes the address of the instruction for which the particular data value is destined, and other information defining the computational context in which that value is used. This context is called the value's *colour*. The data value, together with its tag, is called a *token*. By considering the creation of a token as a single unit of "work", an effective metric for "work" can be derived. For example, where a *proliferate* instruction counts as only one instruction it may create many tokens. To count instructions only would obscure the actual work performed by the system. The number of *Created Tokens* is, therefore, our measure of work.

For complete details of these architectures, the reader is referred to the literature [2, 3, 4] and for a longer version of this paper to [5]. The remainder of this paper contains a summary of the experimental framework, justifying *Created Tokens* as a metric, and a summary of the results obtained from experiments performed using simulators for the various machines.

2 Experimental Framework

The difficulty in comparing systems, as distinct as the MDFM, SDFA and CSIRAC II machines, arises from the need to measure something in common to all systems that represents the amount of work done by each systems. Instruction count is inappropriate since the complexity of the MDFM's and the CSIRAC II's instruction-sets far exceeds that of the SDFA system. Operation count is ruled out as defining what constitutes an operation can be rather difficult, and the use of cycle count assumes that the architecture is independent of technology, clearly a falsehood. Likewise, using execution time is out of the question.

Since all data values in these systems are carried on tokens, the number of tokens generated by all components of the system is taken as a measure of the total work. The creation of a token, as an abstraction of work done by a Data-Flow system, is quite natural, and it is straight forward to identify where a token is created or copied within a Data-Flow system. Notice that this metric includes tokens generated by the MDFM and the CSIRAC II's structure memories as well as by their processing elements. The SDFA system has no structure memories.

Two assumptions underlie the experiments discussed below. First, the CSIRAC II has the ability to transmit multiple values, such as vectors, as a single "token" in the form of a multiple word network packet. For comparison purposes, each such CSIRAC II network word (containing two values) is counted as containing two tokens. Second, the structure memory garbage collection in the CSIRAC II and MDFM is not included in the metric. There are no such overheads for the SDFA or CSIRAC II when using transmitted data structures.

Four benchmark programs were used in this study: Gaussian Elimination (GE), Matrix Multiply (MM), the N-Queens on an N by N Chess Board (NQ), and the Shallow Water Wave Equation (SW). The codes are all available in SISAL from the authors by e-mail and listings can be found in [4].

For the MDFM and CSIRAC II, compiler options were varied individually for each program to achieve the best possible performance, and the SDFA codes were written in a macro assembler, as a compiler is not yet available. For further details and a justification of the "fairness" of these comparisons, see [5].

3 Results

Two comparative experiments were conducted on the three systems. The first compares the total work performed, and the second examines the locality in the three systems.

Total Work: Table 1 contains the results of this experiment[1]. It is clear that the "stateless" computational model (as in SDFA and transmitted structures versions of CSIRAC II codes) carries with it two specialized costs. First, there is the potential that additional copies of data structures might be required that are not required in state based solutions. Second, since "stateless" data structures are passed in their entirety through function boundaries, rather than just the pointers, re-colouring of data structure elements can prove costly.

Because two dimensional arrays are not first class constructs in SISAL[2], there is a penalty for the MDFM and CSIRAC II (in GE & SW). The CSIRAC II suffers from this penalty more than the MDFM (in GE) due to the absence of low level optimizations available on the MDFM.

In NQ, the rapid growth in computational load with board size results in a sudden increase in the number of tokens created. Because of this exponential growth, a small difference in the number of tokens created in the multiply recursive function causes a

[1] The values for NQ(8) on SDFA and CSIRAC and GE(40 & 48) on SDFA are extrapolations.

[2] They are represented as an array of pointers to one dimensional arrays.

Benchmark	CSIRAC	MDFM	SDFA
GE (16)	53	48	41
GE (24)	235	133	131
GE (32)	538	344	302
GE (40)	1030	637	580
GE (48)	1757	1063	992
MM (16)	39	38	38
MM (24)	119	112	120
MM (32)	268	247	274
MM (40)	507	462	522
MM (48)	911	775	887
NQ (4)	11	10	18
NQ (5)	49	42	83
NQ (6)	228	187	400
NQ (7)	1042	823	1833
NQ (8)	4686	3945	8575
SW (4)	102	110	27
SW (8)	315	299	104
SW (12)	639	580	233
SW (16)	1076	952	412
SW (20)	1627	1415	643

Table 1: Tokens Created (in thousands of tokens).

substantial difference in the final token count. This accounts for the failure of the SDFA system to track the other two as closely as it does for the other benchmarks.

Locality: One of the fundamental motivations behind both the SDFA and CSIRAC II architectures was to benefit from the distributed memory model. This experiment compares the amount of work performed locally for each of the three machines. The important part of the global token traffic is that which travels to and from the structure memories, because it is this that will always be global, regardless of what locality facilities are incorporated in the processor.

The results obtained from this experiment are presented in table 2. The percentages represent the fraction of all *Created Tokens* which are transmitted across the network (*i.e.* the percentage of non-local token traffic) and the fraction of all *Created Tokens* which were transmitted to or from the structure memories. These are the tokens that must be transmitted across the network, due to the implied shared memory model of stored data structures.

Benchmark	CSIRAC	CSIRAC	MDFM	MDFM	SDFA
	Global	Memory	Global	Memory	Global
GE (48)	52	52	91	31	13
MM (48)	25	0	86	4	6
NQ (7)	4	0	69	4	6
SW (20)	44	44	82	33	4

Table 2: Percent of all tokens transmitted globally and to or from structure memories.

It is acknowledged that the global traffic in the MDFM is a worst case, due to the distribution function and basic architecture. The structure memory traffic, on the other hand, represents a best case since it assumes that all non structure memory traffic is local.

4 Conclusions

First, it is possible to compare experimental systems with diverse architectures using a meaningful, simple metric which is not rendered useless by the experimental nature of the systems.

Second, although mainstream Data-Flow computing assumes the existence of a specialized structure memory, the provision of such is not a requirement. The stateless operation of the systems above did not cause substantial reductions in performance.

Third, it has been argued recently, see [1], that the increased processor state required to support latency hiding in the traditional Data-Flow systems will eventually force either a limit on the scalability of the system (due to increasing latency) or a reduction in performance (due to the cost context switching). A multiple level memory hierarchy, which can be exploited through locality in computations, reduces the impact of this argument. In two of the systems presented here (SDFA and CSIRAC II), a memory hierarchy exists and the systems employ a variety of techniques to extract locality from the computation. Although there is a barrier inhibiting the scaling of traditional Data-Flow systems, there is no fundamental limit.

Lastly, in the context of evolving work on multi-threaded architectures, the tokens created locally to the processor may be regarded as being equivalent to register transfers, thus providing us with the multi-threaded equivalent of *Tokens Created.* Aggregation of low level instructions into threads is still not well understood, but what seems clear is that in practical codes the larger the multi-processor configuration the smaller the aggregations must be to permit effective load balancing and work distribution. Tokens passing across the network are equivalent in both cases, and as hardware designers, we know that this traffic is an essential parameter in sizing the communication network. Unlike *Tokens Created* or its multi-threaded equivalent, the more conventional measure of instruction count is of almost no use in this critical element of multiprocessor design.

References

[1] D.E. Culler, K.E. Schauser, and T. von Eicken, Two Fundamental Limits in Data-Flow Multiprocessing, Tech. Rept. UBC/CSD 92/716 Computer Science Division. University of California, Berkeley Elsevier (1992).

[2] Egan, G.K., N.J. Webb and A.P.W. Böhm, Some Features of the CSIRAC II Dataflow Machine Architecture, In Advanced Topics in Data-Flow Computing, Prentice-Hall 1991, pp. 143-173.

[3] J.R. Gurd, C. Kirkham, and W. Böhm, The Manchester Dataflow Computing System, In Experimental Parallel Computing Architectures, North-Holland, J.J. Dongarra, Ch. 6, Amsterdahm (1987) pp. 177-220.

[4] Snelling, D.F., The Stateless Data-Flow Architecture, Ph.D. Thesis, Dept. Comp. Sci., Univ. Manchester, Tech. Rep. No. UMCS-93-7-2, (July 1993).

[5] Snelling, D.F. and G.K. Egan, A Comparative Study of Data-Flow Architectures, Dept. Comp. Sci., Univ. Manchester, Tech. Rep. No. UMCS-94-4-3, (April 1994).

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Progress Report on Porting Sisal to the EM-4 Multiprocessor

Andrew Sohn[a], Lingmian Kong[a], and Mistuhisa Sato[b]

[a] Dept. of Computer and Information Science, New Jersey Institute of Technology, Newark, New Jersey 07102-1982, U. S. A. {sohn,kong}@cis.njit.edu

[b] Computer Architecture Section, Electrotechnical Laboratory, 1-1-4 Umezono, Tsukuba-shi, Ibaraki 305, Japan msato@etl.go.jp

Abstract: The functional language Sisal and its compiler OSC are known to provide programmability and performance for shared-memory single-address space multiprocessors. However, their performance on distributed-memory multiprocessors is yet to be investigated. This report presents our on-going efforts of porting Sisal to the 80-processor EM-4 distributed-memory multiprocessor. The key idea of our approach is medium-grain multithreading and simple-minded element-wise data distribution. Explicit-switching based medium-grain threads extracted from Sisal IF2 graph are designed to overlap computation and communication while the element-wise data distribution strategy simplifies data distribution. A runtime system based on an *n*-master-*m*-slave computation model is currently being developed to execute medium-grain threads. Preliminary execution results indicate that the proposed approach is a feasible way of programming distributed-memory machines while providing programmability and performance.

Keyword Codes: C.1.2; D.1.1; D.1.3
Keywords: Distributed-memory multiprocessor; multithreading; functional programming

1 Introduction

Functional languages have proven to provide programmability and performance for shared-memory single-address space multiprocessors. Recent experimental results have indicated that the functional language Sisal [3] and its Optimizing Sisal Compiler [2] yield performance comparable to that of the best Fortran compiler on Crays and other shared-memory machines [1]. However, little practical results have been reported to date on *distributed-memory* multiprocessors. The main difficulties are latency caused by remote memory operations due to data distribution. It is therefore indispensable to develop a programming model which can overcome the difficulties of the distributed-memory machines if they can be scaled to 100 or 1000 processors and to be considered as serious contenders for shared-memory machines. It is precisely the purpose of this research to solve these difficulties.

Specifically, we are currently porting the functional language Sisal to the EM-4 distributed-memory multiprocessor [4,5]. The key idea of our approach is *medium-grain* threads which can support a latency tolerant execution model. We take an optimized IF2 graph [2] which is generated from Sisal and formulate medium-grain threads. Those medium grain threads are then executed on the EM-4 by a runtime system tailored to the EM-4 multiprocessor. The runtime model that we use for executing medium-grain threads is an *n*-master-*m*-slave (*n-m-m-s*) model. The *n-m-m-s* model logically divides all processors into *n* groups according to the network topology. Each group has a master and *m* slaves. It is these *n* masters which access the thread pool for workload distribution. Slave processors access their respective master for obtaining workload. Section 2 presents our approach to programming distributed-memory multiprocessors, including thread generator, an element-wise data distribution policy, and an *n*-master-*m*-slave runtime model for workload distribution. Section 3 lists preliminary experimental results and concludes our progress report.

2 Sisal-EM-4 Environment

Sisal (Streams and Iterations in Single Assignment Language) is a side-effect free functional language [3]. It provides a parallel loop construct which is the main source of parallelism. The Optimizing Sisal Compiler automatically parallelizes and optimizes various parts of Sisal programs for shared-memory machines. Figure 1 shows how OSC translates the input Sisal code to machine executable code.

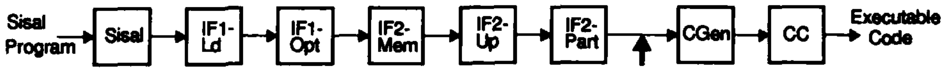


Figure 1: Optimizing Sisal Compiler. IF2mem and IF2Up are array optimizers. IF2Part partitions the data-flow graph for CGen which generates C code for target machine. Our approach (up arrow) starts from IF2Part.

The key idea of our approach to programming distributed-memory machines is the use of medium-grain threads. Figure 2 shows our approach to programming the EM-4 using Sisal, starting from IF2Part.

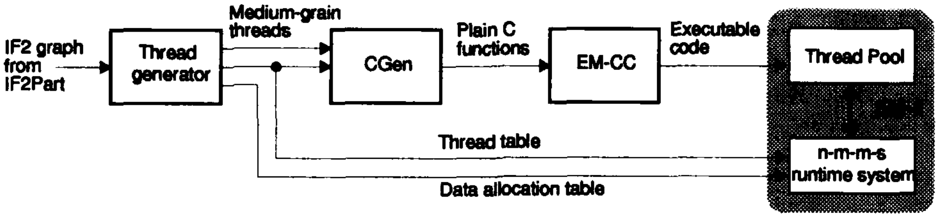


Figure 2: Organization of the Sisal-EM4 environment. CGen is slightly modified for generating threads from IF2 graph. EM-CC is a C compiler for the EM-4, which needs no modification.

Medium-grain thread generator. Given the partitioned IF2 graph, thread generator generates medium-grain threads and other information for data distribution. Parallel Sisal loops are the main source for thread formulation. One of the criteria for thread formulation is the estimated computational cost of each node in the partitioned IF2 graph. The output graph contains nodes, each of which is approximately equal in estimated computational cost. Since OSC focuses only on parallel loops, the codes other than parallel loops are not parallelized. Our effort of formulating threads therefore extends to other nodes, beyond parallel loops. Those IF2 nodes with a large loop iterations will be split into threads each of which will have similar computational cost. Those sequential primitive nodes will form a thread whose estimated computational cost is similar with other threads. The thread formulation method described above has a serious problem. The estimated cost of nodes in IF2 is based on shared-memory machines and will be much different for distributed-memory machines due to data distribution. For this reason, we are currently modifying the computational cost using the element-wise distribution policy.

Element-wise data distribution policy. Data distribution is central to the performance of distributed-memory multiprocessors. If data distribution does not match workload distribution (loop-slice distribution in our case), the number of remote memory operations will increase, thereby resulting in higher communication and synchronization overhead. However, we believe that the effects of data distribution to performance can be reduced by overlapping computation and communication, i.e., multithreading. We therefore adopt a simple-minded *element-wise* data distribution for programmability while its inefficiency resulting from simple distribution will be offset by multithreading. Element-wise data distribution divides data into *n* blocks and allocates a block to each processor, where *n* is a number of processors and a block consists of *consecutive* data elements preserving data locality. Table 1 shows distributing an array of 22 elements to six processors using element-wise consecutive and round-robin methods. Our experimental results indicated that this simple-minded data distribution strategy can indeed give at least 60% performance of the best performing algorithms [6].

Processor number	0	1	2	3	4	5
Round-robin distribution	1,7,13,19	2,8,14,20	3,9,15,21	4,10,16,22	5,11,17	6,12,18
Element-wise distribution	1,2,3,4	5,6,7,8	9,10,11,12	13,14,15,16	17,18,19	20,21,22

Table 1: Distribution of an array of 22-elements to six processors.

***N*-master-*m*-slave runtime model for workload distribution.** The current OSC implementation uses a worker runtime model for shared-memory multiprocessors [2]. We believe that this worker model will be inefficient for distributed-memory machines because it needs a shared global address location where every processor checks to see if there is work to do. Furthermore, a software implementation of global memory on a designated processor will lose 'scalability' which the distributed-memory machines promise to provide. We therefore adopt an *n*-master-*m*-slave runtime model, where there are *n* masters and *m* slaves for each master. By introducing a single level of hierarchy, the model will be able to ease the network congestion and possibly hot spot. Figure 3 shows the 16-master-5-slave model for the 80-processor EM-4.

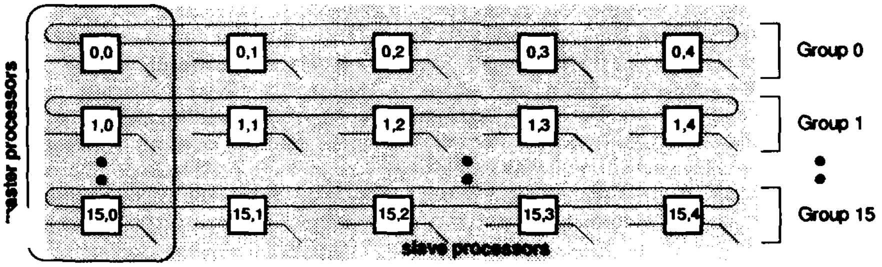


Figure 3: The 16-master-5-slave runtime model for 80 processors. Slaves communicate only to their masters.

3 Preliminary Experimental Results and Discussion

We have executed two livermore loops (loop1 and loop7) and matrix multiply on the EM-4 based on our runtime model and data distribution policy. A simplified version of Loop1 is shown below.

```
loop1: for (j=0;j<n;j++) x[j] = q + (y[j] * (r * z[j+10] + t * z[j+11]));
```

The runtime system is currently being implemented. The results presented in this report are based on hand-coded C functions. Tables below list preliminary experimental results for the three problems.

Matrix multiply: dimension *n*	20	40	60	80	100	120	140	160	180	200
Exec. time on 1 Processor T_1 (sec)	0.058	0.461	1.551	3.668	7.155	12.354	19.607	29.254	41.638	57.750
Exec. time on 80 Procsrs T_{80} (sec)	0.016	0.068	0.158	0.251	0.657	0.965	1.339	1.758	3.072	3.838
Speedup = T_1/T_{80}	3.6	6.8	9.8	14.6	10.9	12.8	14.6	16.6	13.6	15.0

Loop1: Array size *n*	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
Exec. time on 1 Processor T_1 (sec)	0.104	0.208	0.312	0.416	0.520	0.624	0.728	0.832	0.936	1.040
Exec. time on 80 Procsrs T_{80} (sec)	0.005	0.007	0.010	0.013	0.015	0.018	0.021	0.023	0.026	0.029
Speedup = T_1/T_{80}	20.8	29.7	31.2	32.0	34.7	34.7	34.7	36.2	36.0	35.9

Loop7: Array size *n*	2000	4000	6000	8000	10000	12000	14000	16000	18000	20000
Exec. time on 1 Processor T_1 (sec)	0.235	0.470	0.706	0.941	1.176	1.411	1.646	1.882	2.117	2.352
Exec. time on 80 Procsrs T_{80} (sec)	0.007	0.013	0.020	0.026	0.033	0.039	0.046	0.052	0.059	0.065
Speedup = T_1/T_{80}	33.6	36.2	35.3	36.2	35.6	36.2	35.8	36.2	35.9	36.2

Experimental results on matrix multiply are not as promising as expected. The maximum speedup is slightly over 16 on 80 processors, which is nowhere close to commonly reported linear speedup for matrix multiply. The results are based on medium-grain threading. Each thread is a single *i*-loop instance (or *n j*-loop instances), where *n* is matrix dimension. There are several reasons for this modest speedup: First, there are too many remote reads/thread. Second, there are too few threads. Third, there are little number of instructions in each thread compared to remote reads. For *nxn* matrix multiply, a processor has *n/p* threads (in actual implementation some processors have one more thread than others). Each thread has on the average $2n^2$ remote reads and $2n^2$ adds and multiplies, ignoring branch related instructions. The ratio of remote reads to computation is essentially 1, which implies that a remote read must be complete in one instruction cycle in order to tolerate the latency. More discussions are presented in [6].

The two Livermore loops, on the other hand, showed promising results. Their speedup reached over 36-fold on 80 processors. The main reason is the division of loop instances. We divided *n* loop instances into three parts depending on array access: *local-access-only* (LAO), *local-and-remote-access* (LARA), and *remote-access-only* (RAO). LAO part consists of *loop instances* which access array elements allocated to local memory. RAO part consists of loop instances which all require remote reads. LARA part has loop instances which need both local and remote reads. For Loop1 with n=8000 and 100 elements per processor, each of the 80 processors executes 100 iterations. The loop is divided into three parts: LAO with iterations of i=0 to 89, LARA with i=90, and RAO with i=91 to 99. The value 89 of LAO is computed based on the largest array index 11 of Loop1. The LARA loop instance (iteration 89) is unrolled and their remote reads are requested before computation proceeds. *This restructuring of LARA part is the key to the high speedup* since those array elements read in the LARA loop instances are *reused* by RAO instances. Preliminary experimental results indicate that programming distributed-memory machines using Sisal with multithreading would indeed provide programmability and performance for general purpose parallel computing.

Acknowledgments

We would like to thank the EM-4 group members, Yuetsu Kodama, Shuichi Sakai, and Yoshinori Yamaguchi, of the computer architecture section of the Electrotechnical Laboratory (ETL) for providing access to the EM-4. Special thanks go to Hayato Yamana for timely help in executing various programs on the EM-4. We also wish to thank Chinhyun Kim of the University of Southern California for his comments on the *n*-master-*m*-salve runtime model. Andrew Sohn is grateful to Kenji Nishida, ETL, of initiating this collaboration and making arrangements for his stay in Japan. We especially thank anonymous referees for much encouragement and valuable comments on the project. This work is based upon the work supported in part by the NSF CGP Science Fellowship INT-9310972, Japanese Information Science Foundation Fellowship 1992, NJIT SBR No. 421820, and Jean-Luc Gaudiot of the University of Southern California.

References

[1] D. C. Cann, "Retire Fortran: A Debate Rekindled," *Comm. of the ACM* 35, August 1992, pp. 81-89.

[2] J. T. Feo, D. C. Cann, and R. R. Oldehoeft, "A Report on the Sisal Language Project," *Journal of Parallel and Distributed Computing* 10, December 1990, pp.349-365.

[3] J.R. McGraw, S.K. Skedzielewski, S.J. Allan, R.R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas, "Sisal: Streams and Iteration in a Single Assignment Language: Reference Manual version 1.2," Lawrence Livermore Laboratory, Livermore, CA, 1985.

[4] S. Sakai, Y. Yamaguchi, K. Hiraki, and T. Yuba, "An Architecture of a Data-flow Single Chip Processor," in *Proc. of ACM ISCA*, Jerusalem, Israel, May 1989, pp.46-53.

[5] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumura, "Thread-based Programming for the EM-4 Hybrid Data-flow Machine," in *Proc of ACM ISCA*, Gold Coast, Australia, May 1992.

[6] A. Sohn, L. Kong, C. Kim, M. Sato, and S. Sakai, "Multithreading with the EM-4 Distributed-Memory Multiprocessor," *Technical Report*, NJIT CIS-31-94, April 1994.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)

Static vs. Dynamic Strategies for Fine-Grain Dataflow Synchronization

Jonas Vasell

Department of Computer Engineering, Chalmers University of Technology,
S-412 96 Göteborg, Sweden, E-mail: vasell@ce.chalmers.se

Abstract: This paper presents and compares two new strategies for controlling the resources required for synchronization purposes in fine grain data flow execution. The strategies offer different tradeoffs between execution time and synchronization, and can be used to control resource requirements in whole, or parts of, any general recursive program. Measurements of the effects of the strategies on some benchmark programs are also presented.

Keyword Codes: C.1.2, D.1.3.
Keywords: Multiple data stream architectures (multiprocessors); Concurrent programming.

1 Introduction

This work addresses the problem of efficient resource allocation for synchronization purposes in fine-grain data flow execution models.

Traditionally, a major distinction has been made between static and dynamic data flow [2], where static and dynamic refers to the allocation of synchronization resources. Of these, dynamic data flow is best at exploiting parallelism. It was, however, early discovered that unrestricted fine-grain dynamic data flow produces more parallelism than can be exploited by practical implementations. This extra parallelism does not come for free as it requires allocation of synchronization resources. The major problem is caused by reentrant graphs, in which for each invocation, new storage space for each arc and a unique identifier are required. Static data flow is however not always a solution to this problem, partly because it cannot always provide sufficient parallelism, partly because it is not possible to express general recursive computations in static data flow. Other solutions include *k-bounded loops* [1] and *hardware throttles* [4]. The k-bounding technique can however not be applied to general recursive computations. The hardware throttle of the Manchester Data-Flow Machine is a more general mechanism, but has the drawback that it uses a non-local specialized runtime mechanism.

Here we propose two new synchronization resource management strategies intended for general recursive and irregular computations that use combinations of compilation techniques and simple local runtime mechanisms.

2 Proposed Resource Management Strategies

The resource management strategies presented here[1] differ in the restrictions they impose on the order in which different users (function applications) are allowed access to a shared reentrant graph. By varying these restrictions, the different strategies result in data flow graphs that can be placed on a spectrum ranging from *Fully Static Data Flow* (FSDF) to *Fully Dynamic Data Flow* (FDDF). We propose two new strategies that fall between these endpoints. The first is called *Semi-Static Data Flow* (SSDF), and the second (which is "more dynamic") is called *Semi-Dynamic Data Flow* (SDDF).

The access restrictior make a difference between *recursive* and *non-recursive* calls to a shared graph. Recursive calls are calls made from within a shared graph to the same graph. Non-recursive calls are all other calls. We assume that every non-recursive function application in a program results in a unique copy of the function graph. Since the non-recursive application can be executed several times if it is part of a recursive function, several non-recursive calls to a shared graph can be generated, but only from one single point. We will also assume that all recursive applications of a function in a program are given unique indexes called *recursion levels.* All other expressions are assigned a recursion level by taking the maximum recursion level of its subexpressions or zero if it has no subexpressions.

In FSDF, SSDF, and SDDF, non-recursive calls are not allowed to access a shared graph before the previous non-recursive call to the same graph has produced a result and terminated. In FDDF, non-recursive calls are allowed to access a graph at any time.

FSDF does not allow a recursive call until the previous recursive call has produced a result and terminated. The effect of this restriction is that FSDF only allows tail-recursive functions or loops. SSDF allows the recursive call with the lowest recursion level to access the graph at any time. A recursive call C with a higher recursion level is only allowed to access the graph when all recursive calls with lower recursion levels that were generated in parallel with C have terminated. This allows all forms of recursion, but limits the parallelism. SDDF and FDDF allow all recursive calls to be executed at any time.

The above gives that for all strategies except FSDF, more than one call can be using a shared graph at the same time. This makes it necessary to distinguish arguments and partial results that belong to different calls by assigning *tags.* Tags are allocated when a call enters a shared graph. When the call exits the graph, the tag is deallocated (made free for use by another call) and the tag that the arguments of the call had before entering the graph is assigned to the result.

In SSDF and SDDF, a non-recursive call is given tag zero. Each non-recursive call in FDDF is given a unique tag that is not being used in the shared graph when the call is made.

Each call to a shared graph for a recursive function can cause one or more recursive calls. In SSDF, each recursive call is given a new tag by adding one to the tag of the call that generated it. With SDDF, local tag allocation is still possible, but it is a little more complex than in SSDF. The new tag is computed by multiplying the old tag with the maximum number of recursive calls that can be started concurrently and adding the recursion level of the recursive call. A disadvantage with this tag allocation method is that it can leave unused "holes" in the tag space for irregular functions which make varying numbers of recursive calls, and it does not allow any tag value to be used more than once for each non-recursive application. This tag allocation technique will not work for FDDF,

[1] Only a brief summary is presented here. A more detailed description is available as a technical report.

which instead has to rely on a more expensive or slower global tag allocator.

In SSDF, SDDF, and FDDF, we have chosen to make the synchronization operations explicit and to avoid using them where they are not needed. Thus, we use two kinds of nodes. *Ordinary nodes* (*e.g.* operators and switches) operate according to the static data flow firing rule, *i.e.*, they will produce a results when input data are available at the inputs and the output arcs are empty. *Synchronization nodes* have a synchronization memory which either stores incoming data from two inputs and outputs matching pairs of data with the same tag (*wait-nodes*), or acts as a FIFO buffer for incoming data (*wait-buffers*).

As mentioned above, wait-nodes are required at the input of all operators that depend on results from expressions with different recursion levels. However, in SSDF the order in which the inputs to such nodes will be computed is known, and thus the synchronization can be made quicker than in SDDF and FDDF. This will give SSDF an advantage which can be seen in some of the results in Section 3.

In SSDF, wait-nodes are also used to prevent arguments to recursive calls to enter the shared graph until all calls with the same tag and a lower recursion level have terminated. In both SSDF and SDDF, simplified wait-nodes which do not require any memory are used to stop more than one non-recursive call from using a shared graph at any time. In SDDF and FDDF, wait-buffers are needed at the argument entry points of a shared graph to prevent dead-lock.

3 Results and Conclusions

Comparisons between the different strategies have been made using a dataflow compiler and simulator developed according to the basic principles described in Section 2, for a subset[2] of the Id dataflow language [3]. The simulator assumes an unlimited number of processors, single clock cycle execution time for all nodes, zero communication delay over all arcs, and zero data and synchronization memory access delay. It also assumes that tag allocation and deallocation takes no time, which favors the FDDF strategy.

A set of benchmark programs have been compiled according to the SSDF, SDDF, and FDDF strategies. Through simulations, the effects of the three different strategies on execution time and required synchronization resources have been measured. FSDF is not included as it does not allow general recursion. As our primary interest is to investigate resource management strategies for irregular, general recursive functions, rather than functions based on regular loops, the benchmarks fall primarily in the former category. *Primes* computes a list of prime numbers up to a given limit. *Sort* sorts a list of numbers using the insertion sort algorithm. *Queens* finds a solution to the Eight Queens problem. *Find* searches for the first, longest occurrence of a pattern in text. The pattern can contain wildcard characters. *Fib* computes the n:th Fibonacci number. *Evdist* computes the "evolutionary distance" between two character strings, *i.e.*, the minimum cost of transforming one string to the other using fixed cost insert and delete operations.

The results for the SDDF and FDDF strategies are presented in Table 1. For each benchmark, program execution time, the number of synchronization nodes, and the total required synchronization memory (the sum of used memory in all synchronization nodes) are presented. All numbers are relative to the SSDF strategy.

The results show that for all of the programs, SSDF requires less synchronization resources than both SDDF and FDDF. In terms of execution time, the results can be

[2] The subset excludes higher order functions, list and array comprehensions, and user defined data types (but includes basic scalar types, lists, one- and two-dimensional arrays).

Bench-mark	SDDF			FDDF		
	Exec time	Sync. nodes	Sync. mem.	Exec time	Sync. nodes	Sync. mem.
Queens	1.59	6.30	3.58	1.71	7.15	7.84
Find	1.66	5.04	6.27	1.71	5.41	7.25
Sort	1.35	3.45	3.37	0.40	4.09	10.83
Primes	1.25	2.65	3.31	0.67	3.13	8.24
Fib	0.17	2.75	347.92	0.15	2.88	63.08
Evdist	0.20	3.47	153.14	0.20	3.59	51.72

Table 1: Results.

divided into three categories. In the first category, Queens and Find, the most efficient execution is achieved with the SSDF strategy. This indicates that these programs have little overlap between both non-recursive calls and recursive calls of functions, and that they gain from the lower synchronization overhead of SSDF. The second category consists of Sort and Primes, which gain from FDDF but not from SDDF. This is consistent with the fact that these programs make frequent repeated non-recursive calls, which can be overlapped in FDDF. As in the first category, SDDF loses against SSDF because of the higher synchronization overhead. In the last category are Fib and Evdist, which are branching recursive and thus can exploit the overlap of recursive calls offered by SDDF and FDDF but not SSDF. These programs also show the inefficiency of the local tag allocation technique used in SDDF.

With these results, we have shown that due to its low synchronization overhead, SSDF can sometimes offer a useful solution to the synchronization resource management problem. It can be an alternative to k-bounded loops and hardware throttling in cases where general recursion is required and specialized non-local resource managers have to be avoided. Its primary disadvantage is that it is not capable of adapting to run-time load variations. Due to its inefficient use of synchronization memory and its lack of performance gain over FDDF, SDDF appears to have no advantages. However, the effects of tag allocation overhead in FDDF have not been taken into account.

References

[1] D. E. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings 15:th Annual Symposium on Computer Architecture*, pages 141-150, 1988.

[2] J.-L. Gaudiot and L. Bic, editors. Advanced Topics in DataFlow Computing. Prentice Hall, 1991.

[3] R. S. Nikhil. The parallel programming language Id and its compilation for parallel machines. CSG Memo 313, Computation Structures Group, LFCS, Massachusetts Institute of Technology, 1990.

[4] C. A. Ruggiero and J. Sargeant. Control of parallelism in the Manchester data-flow computer. In *Lecture Notes in Computer Science*, No. 274, pages 1-15, 1987.

Parallel Architectures and Compilation Techniques (A-50)
M. Cosnard, G.R. Gao and G.M. Silberman (Editors)
Elsevier Science B.V. (North-Holland)
© 1994 IFIP. All rights reserved.

Trace Software Pipeli[illegible] A Novel Technique for Parallelization of Loops with Branches

Jian Wang[a], Andreas Krall[a], M. Anton Ertl[a] and Christine Eisenbeis[b]

[a]Institut für Computersprachen, Technische Universität Wien, Argentinierstr. 8, A-1040 Wien, Austria. Email: jian@mips.complang.tuwien.ac.at

[b]INRIA-Rocquencourt, Domaine de Voluceau, BP 105-78153, Le Chesnay Cedex, France

Abstract: Trace software pipelining is a novel global software pipelining technique. It can exploit instruction-level parallelism across all iterations of a loop by compacting the original loop body with any global loop-free code scheduling technique. The resulting loop is called a trace software pipelined (TSP) code, which can be executed directly with a special architectural support or be transformed into a globally software pipelined loop for the current VLIW and superscalar processors.

Keyword Codes: D.1.3; D.2.2
Keywords: Concurrent Programming; Tools and Techniques

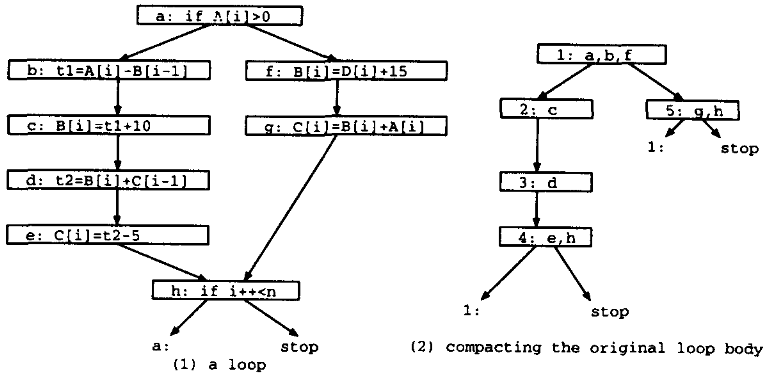
1 Introduction

Global software pipelining is a sophisticated but efficient compilation technique to exploit Instruction-Level Parallelism (ILP) for loops with branches [1,2,3,4,5]. Trace Software Pipelining (TSP) is a novel technique for performing global software pipelining, which can exploit ILP across all iterations of a loop by compacting the original loop body while ignoring the constraint of some data dependences.

After full renaming, we can remove all anti-dependences in the example of Fig.1(1). There are loop-independent dependences between b and c, c and d, d and e, f and g. There are loop-carried dependences between c and b, e and d, f and b, g and d. Under the constraint of these data dependences, we can apply a global code scheduling technique to compact the original loop body (see Fig.1(2)), but we can only exploit the ILP within the loop body and can not exploit the ILP across iterations.

The idea behind TSP is that we compact the original loop body with ignoring the constraint of some data dependences. For example, after we ignore the constraint of the data dependences between c and d, f and g, we can globally compact the loop body to get the compacted code, called TSP code, shown in Fig.2(1).

It is not easy to understand the TSP code before we introduce a new concept, *iteration-number*. In order to preserve the program semantics, those ignored data dependences should be recovered in the final pipelined loop. Therefore, it is required that the operations in the TSP code come from different iterations. We attach an iteration-number to each operation. In Fig.2(1), the iteration-numbers of a,b,c,h and f are 2 and those of d,e

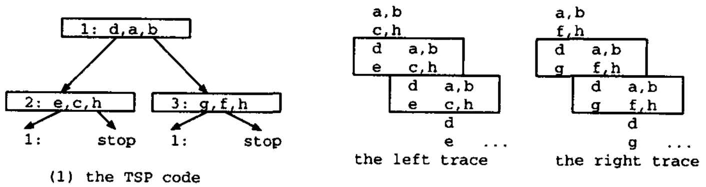


(1) a loop

(2) compacting the original loop body

Fig.1 An Example

and g are 1. That means, if d,e and g come from $(i+1)$th iteration, then a,b,c,h and f come from $(i+2)$th iteration. With the iteration-numbers, we can view the TSP code as a unified form of the pipelining of all traces[1] (see Fig.2(2)).



(1) the TSP code

(2) the pipelining of each trace

Fig.2 TSP Code

Thus, we have two problems to be addressed in this paper. (1) How to choose the ignored dependences to derive a TSP code (Section 2)? (2) How to execute the TSP code (Section 3)?

2 TSP Code

During compaction of the loop body, not all data dependences can be ignored. For the loop in Fig.1(1), we can ignore the constraint of the dependences between c and d, f and g, but we can not ignore the dependences between b and c, d and e.

The data dependence between operation op_i and op_j can be represented by two non-negative integers, $\lambda(op_i, op_j)$ and $\delta(op_i, op_j)$. For example, let (op_i, op_j) be a data dependence, then op_j can only be executed $\delta(op_i, op_j)$ cycles after op_i of the $\lambda(op_i, op_j)$th previous iteration has started executing [3,6].

Definition 1: Let L be a loop, $TSP(L)$ be its TSP code. $TSP(L)$ is a valid trace software pipelined code of L if and only if we can find an iteration-number for each operation of L, such that the following two conditions are satisfied:
(1) Let $itn(op)$ be the iteration-number of op, $dis(op_i, op_j)$ be the length of the path from

[1] We define a *trace* as a path from the entrance to one of the exits in the loop body.

op_i to op_j, $len(p)$ be the length of the trace p. Then for any trace p of $TSP(L)$ and any pair of operations (op_i, op_j) of p with a data dependence, $(itn(op_j) - itn(op_i) + \lambda(op_i, op_j)) * len(p) + dis(op_i, op_j) \geq \delta(op_i, op_j)$.
(2) Let op_t be a test operation, $B1$ and $B2$ are its branches in $TSP(L)$, for any $op \in B1 \cup B2$, if op is not one of the operations which are moved down across op_t then $itn(op_t) \geq itn(op)$; $\triangleleft$

The first condition guarantees that all ignored dependences can be recovered and the second guarantees that we can get a transformed sequential loop (see the next section).

We present a three-step method[2] for constructing a TSP code: (1) Find out the strongly connected components; (2) Select those loop-independent dependences not included in the strongly connected components as the dependences whose constraints can be ignored; (3) Call a global code scheduling technique to globally compact the original loop body with ingorance of the constraint of those selected loop-independent dependences. We have proven that the method can generate a valid TSP code. The iteration-numbers of all operations can be computed by Definition 1. Wang and Eisenbeis give a detailed method in [6].

3 Execution of TSP Code

3.1 Hardware Method

A TSP code can be executed directly with a special architectural support. The principle is that, when the loop continuously executes a trace, the pipelining of the trace is executed; when the loop transfers from a trace to another, the postlude of the former and the prelude of the latter should be first executed and then the pipelining of the latter is executed. Hence, a trace analyzer is needed to check the executing trace in run-time.

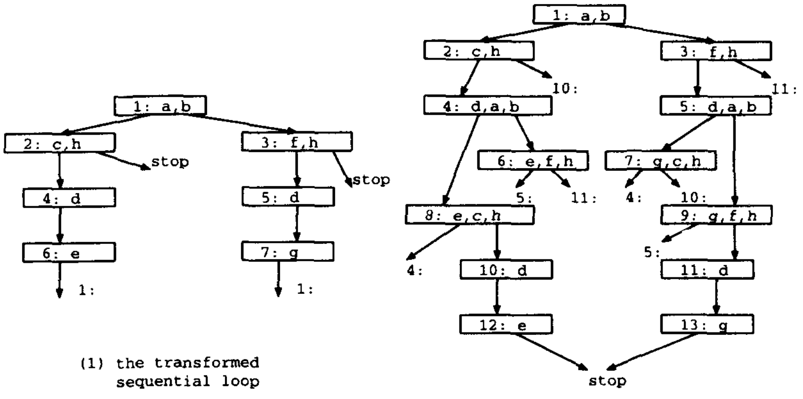
3.2 Software Method

A TSP code can be transformed at compile time into an equivalent globally software pipelined loop. As the first step, we transform the TSP code into a sequential loop which can be executed in the way of preserving the program semantics. For example, the sequential loop transformed from the TSP code in Fig.2(1) is shown in Fig.3(1). In the transformed sequential loop, we only exploit the ILP within the loop body; and the ILP across loop bodies will be exploited in the next step. Next, we pipeline the transformed sequential loop bodies. The TSP code gives two constraints on the pipelining process: (1) It gives the initiation intervals for all loop traces; and (2) it gives the pipelined results for all loop traces. Thus, the pipelining is easily done. It is only necessary to schedule the "transfering" postludes and preludes. For the example in Fig.3(1), the final result is shown in Fig.3(2).

4 Conclusion

Trace software pipelining expresses the globally software pipelined loop in the form of TSP code, thereby exploiting ILP across all iterations of a loop can be done by compacting the original loop body with a global loop-free code scheduling technique. A TSP code can be

[2] This method is efficient if full renaming is done.



(1) the transformed sequential loop

(2) an equivalent globally software pipelined loop

Fig.3 Transformation from TSP Code to Globally Software Pipelined Loop

executed directly with an architectural support, so we expect that trace software pipelining is promising for solving the code complexity problem of global software pipelining.

Acknowledgements

This work was mainly supported by the Lise Meitner Stipendium funded by the Austrian Science Foundation (FWF) and the Austrian Science and Research Ministry and had in part begun when Jian Wang was a post-doc researcher in INRIA. We would like to thank Prof. Manfred Brockhaus for his helpful comments.

References

[1] A. Aiken and A. Nicolau, A Realistic Resource-Constrainted Software Pipelining Algorithm, Languages and Compilers for Parallel Computing, A. Nicolau, D. Gelernter, T.Gross and D. Padua (Editor), Pitman/The MIT Press, London, 1991, 274-290.

[2] K. Ebcioglu and T. Nakatani, A New Compilation Technique for Paralelizing Loops with Unpredictable Branches on a VLIW Architecture, Languages and Compilers for Parallel Computing, D. Gelernter, A. Nicolau and D. Padua (Editor), Pitman/The MIT Press, 1989, 213-229.

[3] M.S. Lam, A Systolic Array Optimizing Compiler, Ph.D. Thesis, CMU-CS-87-187, 1987.

[4] B. Su, S. Ding, J. Wang and J. Xia, GURPR–A Method for Global Software Pipelining, Proc. of MICRO-20, 1987.

[5] Bogong Su and Jian Wang, *GURPR**: A New Global Software Pipelining Algorithm, Proc. of MICRO-24, 1991.

[6] Jian Wang and Christine Eisenbeis, Decomposed Software Pipelining: A New Approach to Exploit Instruction-Level Parallelism for Loop Programs, Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, edited by M. Cosnard, K. Ebcioglu and J. Gaudiot, North-Holland, 1993, pp.3-14.

IFIP

The INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING is a multinational federation of professional and technical organisations (or national groupings of such organisations) concerned with information processing. From any one country, only one such organisation – which must be representative of the national activities in the field of information processing – can be admitted as a Full Member. In addition a regional group of developing countries can be admitted as a Full Member. On 1 October 1993, 44 organisations were Full Members of the Federation, representing 65 countries.

The aims of IFIP are to promote information science and technology by:

- fostering international co-operation in the field of information processing;
- stimulating research, development and the application of information processing in science and human activity;
- furthering the dissemination and exchange of information about the subject;
- encouraging education in information processing.

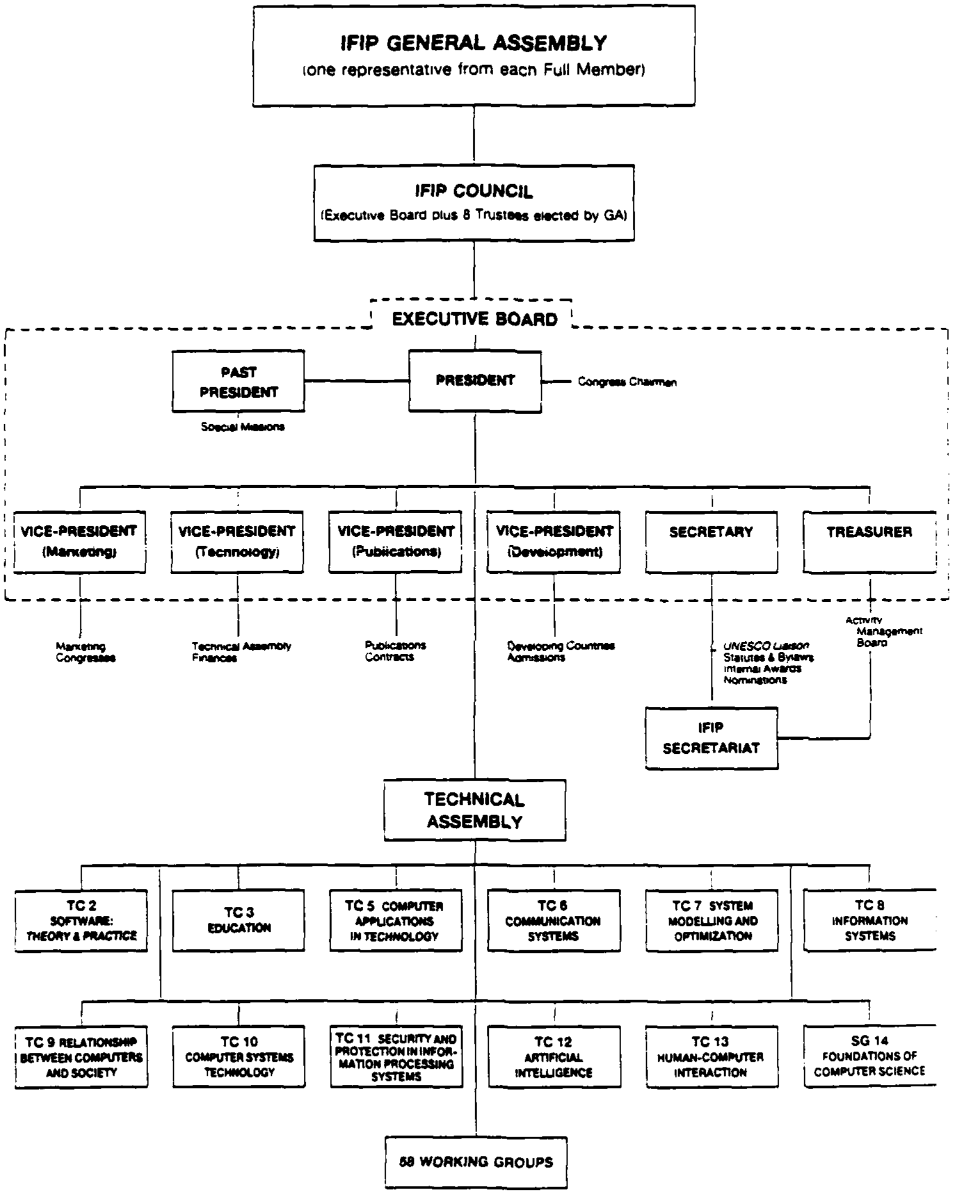
IFIP is dedicated to improving worldwide communication and increased understanding among practitioners of all nations about the role information processing can play in all walks of life.

Information technology is a potent instrument in today's world, affecting people in everything from their education and work to their leisure and in their homes. It is a powerful tool in science and engineering, in commerce and industry, in education and adminstration. It is truly international in its scope and offers a significant opportunity for developing countries. IFIP helps to bring together workers at the leading edge of the technology to share their knowledge and experience, and acts as a catalyst to advance the state of the art.

IFIP came into official existence in January, 1960. It was established to meet a need identified at the first International Conference on Information Processing which was held in Paris in June, 1959, under the sponsorship of UNESCO.

Organisational Structure
The Federation is governed by a GENERAL ASSEMBLY, which meets once every year and consists of one representative from each Member organisation. The General Assembly decides on all important matters, such as general policy, the programme of activities, admissions, elections and budget.

IFIP ORGANISATION CHART


IFIP GENERAL ASSEMBLY
(one representative from each Full Member)
IFIP COUNCIL
(Executive Board plus 8 Trustees elected by GA)
EXECUTIVE BOARD
PAST PRESIDENT
Special Missions
PRESIDENT
Congress Chairmen
VICE-PRESIDENT (Marketing)
VICE-PRESIDENT (Technology)
VICE-PRESIDENT (Publications)
VICE-PRESIDENT (Development)
SECRETARY
TREASURER
Marketing
Congresses
Technical Assembly
Finances
Publications
Contracts
Developing Countries
Admissions
UNESCO Liaison
Statutes & Bylaws
Internal Awards
Nominations
Activity
Management
Board
IFIP SECRETARIAT
TECHNICAL ASSEMBLY
TC 2 SOFTWARE: THEORY & PRACTICE
TC 3 EDUCATION
TC 5 COMPUTER APPLICATIONS IN TECHNOLOGY
TC 6 COMMUNICATION SYSTEMS
TC 7 SYSTEM MODELLING AND OPTIMIZATION
TC 8 INFORMATION SYSTEMS
TC 9 RELATIONSHIP BETWEEN COMPUTERS AND SOCIETY
TC 10 COMPUTER SYSTEMS TECHNOLOGY
TC 11 SECURITY AND PROTECTION IN INFORMATION PROCESSING SYSTEMS
TC 12 ARTIFICIAL INTELLIGENCE
TC 13 HUMAN-COMPUTER INTERACTION
SG 14 FOUNDATIONS OF COMPUTER SCIENCE
58 WORKING GROUPS

The day-to-day work of IFIP is directed by its Officers: the President, Vice-Presidents, Secretary and Treasurer, who are elected by the General Assembly and together constitute the EXECUTIVE BOARD.

The COUNCIL, consisting of the Officers and up to eight Trustees elected from the General Assembly, meets twice a year and takes decisions which become necessary between General Assembly meetings.

The headquarters of the Federation are in Geneva, Switzerland where the IFIP Secretariat administers its affairs.

For further information please contact:

IFIP Secretariat
attn. Mme. GWYNETH ROBERTS
16 Place Longemalle
CH-1204 Geneva, Switzerland
telephone: 41 (22) 310 26 49
facsimile: 41 (22) 781 23 22
Bitnet: ifip@cgeuge51

IFIP's MISSION STATEMENT

IFIP's mission is to be the leading, truly international, apolitical organisation which encourages and assists in the development, exploitation and application of Information Technology for the benefit of all people.

Principal Elements

1. To stimulate, encourage and participate in research, development and application of Information Technology (IT) and to foster international co-operation in these activities.
2. To provide a meeting place where national IT Societies can discuss and plan courses of action on issues in our field which are of international significance and thereby to forge increasingly strong links between them and with IFIP.
3. To promote international co-operation directly and through national IT Societies in a free environment between individuals, national and international governmental bodies and kindred scientific and professional organisations.
4. To pay special attention to the needs of developing countries and to assist them in appropriate ways to secure the optimum benefit from the application of IT.
5. To promote professionalism, incorporating high standards of ethics and conduct, among all IT practitioners.
6. To provide a forum for assessing the social consequences of IT applications; to campaign for the safe and beneficial development and use of IT and the protection of people from abuse through its improper application.

7. To foster and *facilitate co-operation* between academics, the IT industry and governmental bodies and to seek to represent the interest of users.
8. To provide a vehicle for work on the international aspects of IT development and application including the necessary preparatory work for the generation of international standards.
9. To contribute to the formulation of the education and training needed by IT practitioners, users and the public at large.

IFIP TRANSACTIONS

IFIP TRANSACTIONS is a serial consisting of 15,000 pages of valuable scientific information from leading researchers, published in 36 volumes per year. The serial includes contributed volumes, proceedings of the IFIP World Conferences, and conferences at Technical Committee and Working Group level. Mainstream areas in the IFIP TRANSACTIONS can be found in Computer Science and Technology, Computer Applications in Technology, and Communication Systems.

From 1993 onwards the IFIP TRANSACTIONS are only available as a full set.

IFIP TRANSACTIONS A:
Computer Science and Technology
1992: Volumes A1–A19
1993: Volumes A20–A40
1994: Volumes A41–A61
ISSN 0926-5473

IFIP Technical Committees that are involved in IFIP TRANSACTIONS A
Software: Theory and Practice (TC2)
Education (TC3)
System Modelling and Optimization (TC7)
Information Systems (TC8)
Relationship Between Computers and Society (TC9)
Computer Systems Technology (TC10)
Security and Protection in Information Processing Systems (TC11)
Artificial Intelligence (TC12)
Human-Computer Interaction (TC13)
Foundations of Computer Science (SG14)

IFIP TRANSACTIONS B:
Applications in Technology
1992: Volumes B1–B8
1993: Volumes B9–B14
1994: Volumes B15–B19
ISSN 0926-5481

IFIP Technical Committee that is involved in IFIP TRANSACTIONS B
Computer Applications in Technology (TC5)

IFIP TRANSACTIONS C:
Communication Systems
1992: Volumes C1–C8
1993: Volumes C9–C16
1994: Volumes C17–C26
ISSN 0926-549X

IFIP Technical Committee that is involved in IFIP TRANSACTIONS C
Communication Systems (TC6)

IFIP TRANSACTIONS FULL SET: A, B & C
1992: 35 Volumes. US $ 1892.00/Dfl. 3500.00
1993: 35 Volumes. US $ 2100.00/Dfl. 3885.00
1994: 36 Volumes. US $ 2277.00/Dfl. 4212.00
The Dutch Guilder prices (Dfl.) are definitive. The US $ prices mentioned above are for your guidance only and are subject to exchange rate fluctuations. Prices include postage and handling charges.

The volumes are also available separately in book form.

Please address all orders and correspondence to:

ELSEVIER SCIENCE B.V.
attn. PETRA VAN DER MEER
P.O. Box 103, 1000 AC Amsterdam
The Netherlands
telephone: 31 (20) 5862602
facsimile: 31 (20) 5862616
email: p.meer@elsevier.nl