(2)

AD-A254 554

Program Structure as a Basis for the Parallelization of Global Compiler Optimizations

Angelika Zobel
May 15, 1992
CMU-CS-92-137

DTIC
ELECTE
AUG 17 1992
S B D

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pa 15213-3890

submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science at Carnegie Mellon University

CLEARED
FOR OPEN PUBLICATION

JUL 20 1992 4

[illegible] FREEDOM OF INFORMATION
[illegible] REVIEW [illegible]
DEPARTMENT OF DEFENSE

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

REVIEW OF THIS MATERIAL DOES NOT IMPLY
DEPARTMENT OF DEFENSE INDORSEMENT OF
FACTUAL ACCURACY OR OPINION.

© 1992 by Angelika Zobel

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the Intold Corporation or the U.S. government. This thesis was supported in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330. Work furnished in connection with this research is provided under prime contract MDA972-90-C-0035 issued by DARPA/CMO to Carnegie Mellon University and under its subcontract, No. 334918-58792 with Networks Systems Corporation.

423 887 144p

92-22843

92 8 13 028

92-3110

Keywords: parallel compilation, parallel algorithms, machine independent compiler optimizations

Carnegie
Mellon

School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

Program Structure as a Basis for the
Parallelization of Global Compiler Optimizations

ANGELIKA ZOBEL

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

Thomas Groß ________________ MAJOR PROFESSOR 5/15/92 ________ DATE

Raj Reddy ________________ DEAN 5/15/92. ________ DATE

APPROVED:

________________ PROVOST 3 June 1992 ________ DATE

Abstract

Optimizing compilers can produce very efficient code but incur a high compilation cost. Because interactions between arbitrary points in a program are possible, global compiler optimizations are inherently expensive and in general there is no easy way to partition the data processed during global compiler optimizations into independent components.

This thesis explores how program structure can be used to provide a basis for the parallelization for global compiler optimizations. Two concepts to obtain data parallelism in global compiler optimizations are described. The first concept uses program structure explicitly for the parallelization. demonstrated by the parallelization of interval analysis of global data flow equations. The second concept consists of using program structure analytically to establish data partitioning points. The effectiveness of this concept is demonstrated in the parallelization of global register allocation via optimal coloring of a graph denoting register conflicts, an NP complete problem. A model for global register allocation in which program structure is used to analyze a register conflict graph is presented. The purpose of this analysis is to detect clique separators that partition a conflict graph into independent components that can be colored independently and combined to an overall coloring by renaming. Properties of live ranges in loops and conditionals are linked to characteristics of the conflict graph. If certain restrictions are met by the live ranges that occur in conditionals and loops, the register conflict graph can be transformed to an equivalent interval graph. Interval register conflict graphs are desirable because they can be colored optimally in polynomial time and because *all* clique separators of an interval graph can be located systematically. The experimental evaluation of my method shows that in many cases the entire conflict graph or large portions thereof can be mapped to equivalent interval graphs. Consequently, in such conflict graphs almost all clique separators can be detected which makes it easy to partition the conflict graph into components that can be processed independently. The knowledge about interval portions of register conflict graph can be used both as a platform for the parallelization of global register allocation and to improve sequential register coloring algorithms.

DTIC QUALITY INSPECTED 5

Accession For
NTIS GRA&I ☑
DTIC TAB □
Unannounced □
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

Acknowledgements

I would like to thank everyone who has helped me along this long and enjoyable path. Thomas Gross has provided much support, technical feedback and constructive suggestions throughout my career as a graduate student. I am particularly grateful for his continuous encouragement and the weekly discussions during the course of my thesis work. I would like to thank Mario Barbacci for invaluable discussions and support. Peter Steenkiste has spent many hours of techical discussions with me. I would also like to thank David Wortman for serving as external reader on my thesis committee. Thanks to Chang-Hsin Chang for providing me with the interface to the IWARP compiler, and thanks to David Applegate for his continuous support during the six years we shared the same office. I would also like to express my appreciation to all those that have made CMU SCS such an enjoyable community, and most of all I would like to thank Sharon Burks for her guidance and help throughout the years. I am grateful to Bernd for encourageing me when I needed it and to my father and mother, for the care and support they have given me through all the years. I dedicate this thesis to them.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Optimizing compilers can produce very efficient code but incur a high compilation cost. To hide the details of complicated architectures from the user and at the same time take advantage of a machine's processing power, compilers for such machines must optimize extensively. Compiler optimizations consist of machine independent optimizations that are useful for every backend and machine dependent optimizations tailored to specific architectures. Global compiler optimizations are inherently expensive because they potentially depend on the flow of data in any part of a program. The name suggests that optimizing compilers produce optimal code, and given a specific input program and a specific machine, theoretically it is possible to produce optimal code. In practice however, optimal code can hardly be achieved. The reason is that a number of individual global compiler optimizations are NP complete, so finding an optimal solution for one such optimization can require exponential compilation time. In addition, compiler optimizations interact with each other and therefore optimization times are non trivial even when the resulting code is not optimal. For some optimizing compilers, compilation times measured in hours are not unusual [GrossZobel 89]. This is not surprising when one considers the hard problems that must be solved for example by a compiler for pipelined machines [Lam 88], machines with complicated memory organizations or vectorized machines [Sites 78a, Sites 78b, KuckKuhnEtAl 81]. The trend towards high compilation times is even more evident in compilers for MIMD machines: compilers for MIMD machines must generate efficient code for a number of (potentially different) processors. Ultimately, compilers will be one of the major bottlenecks in the software development cycle.

1.1. Motivation

The focus of this thesis is to investigate parallel frameworks for global compiler optimizations. We chose to investigate the parallelization of global compiler optimizations for two reasons. The initial motivation was the need to speedup compilation. In most optimizing compilers, global optimizations and code generation account for most of the compilation time, so to speedup the compilation process most effectively, the parallelization of global optimizations and code generation must be considered. Unlike code generation, most global optimizations are independent of the target machine, so their parallelization can be incorporated into any optimizing compiler.

The second motivation was to study the interactions between program structure and the characteristics of global optimization problems. In a mathematical sense, many global optimization problems are NP complete. One reason why some global optimization problems are so hard to solve is that the data for such optimizations is processed independently of the input program. One example of such an optimization is global register allocation: a widely used approach to global register allocation is based on graph coloring. For a given input program, a graph denoting register conflicts is constructed and an optimal assignment of registers to live ranges in the program is equivalent to an optimal coloring of this graph. Finding a parallel framework in which it is possible to partition this graph into independent components that can be colored individually is hard when only the graph is considered. In this thesis, we examine whether knowledge about the structure of the input program can be used to detect locality in the data processed in global compiler optimizations that can be used to detect data parallelism.

1.2. Paradigms for parallelism

The parallelization of an algorithm is useful only if certain conditions are met. First, given the solution of sequential algorithm for a problem, the solution of the parallel algorithm for the same problem must be at least as good as the sequential solution. In other words, for the purpose of this thesis we do not accept decrease of the quality of the parallel solution in return for parallel speedup. Second, it must be ensured that partial results can be combined to a global result without compromising the quality of the global result. If for instance the solutions for small components are optimal, we require that the combination of the individual optimal solutions results in an overall optimal solution. Third, the re-combination of partial results to an overall result must be fast enough such that parallel speedup is not offset by the post processing time required for re-combination.

1.3. Benefits of data parallelism

The research for the parallelization of a given problem consists of developing an algorithm that partitions the data into components that can be processed independently. The parallelization of a given problem requires the careful analysis of the problem; in many cases this leads to a better understanding of the problem itself which can lead to an improvement of sequential algorithms as well. Partitioning the data into independent components has advantages for both parallel and sequential implementations, because smaller subcomponents of the input are processed independently.

Divide and conquer:

Given polynomial running time for a sequential algorithm, partitioning the data into independent smaller subcomponents allows the reduction of the overall running time even when the algorithm runs sequentially. Because polynomial running time (at least $O(n^2)$ where n is the size of the input) grows non-linearly with increasing problem size, solving smaller problems is cheaper than solving larger problems. This advantage becomes even more apparent when NP complete problems are considered: dividing a problem into smaller components has the result that without increasing the overall running time, more effort can be spent to find a good solution for the individual components; this means that expensive heuristics or even exhaustive search can be applied if the subproblems can be made small enough.

Parallel speedup:

Given the points at which the data can be partitioned, the parallel speedup depends on several factors:

1. *An upper bound for the theoretical parallel speedup*: The maximal number of processors that *can* operate concurrently for a given problem is an upper bound for the theoretical parallel speedup.
2. *Upper bounds for the observed parallel speedup*: For a given parallelization, the total sequential processing time divided by the maximal processing time of the parallel tasks is an upper bound for the maximal parallel speedup that can be observed. Both the amount of communication between concurrent processes, the time to setup the parallel execution and the postprocessing time limit the actual observed parallel speedup. Systems overhead such as bus contention, scheduling overhead, parallel startup time etc. are problem independent factors that limit the observed parallel speedup.

A successful parallelization of an algorithm therefore consists of two parts, the data partitioning of the input and the parallelization itself.

1.4. Data parallelism based on program structure

Global compiler optimizations are inherently hard to parallelize because potential interactions between *any* two parts of the input program are possible. Due to these interactions, the data that must be processed in global optimizations is highly interconnected and there is no simple way to detect data partitioning points. The research focus of this thesis is to establish the role of program structure for the partitioning of data that is processed in global compiler optimizations. We are more interested in developing frameworks in which data partitionings can be found in a methodic way, and the thesis claim is that this method to find data parallelism is based on the structure of the input program.

1.4.1. Two types of global compiler optimizations

We distinguish between two types of global compiler optimizations: structured and unstructured compiler optimizations. In structured global compiler optimizations the basic blocks of a program are processed in a particular order that is based on the parse tree of the input program. Examples of structured compiler optimizations are optimizations that operate based on a program's loop structure such as vectorization and software pipelining and all data flow problems. Data flow problems can be solved in polynomial time, and there exist several algorithms that take advantage of locality of the data in loops: the partial results for an inner loop need not be recomputed for the data flow analysis of the enclosing loop. Note that in this type of optimization algorithms, backtracking across loop boundaries is usually not applied.

The difference between unstructured and structured optimizations is that in unstructured optimizations the order in which the data of the input program is processed is independent of the parse tree. In general data partitioning based on the parse tree of the input program does not yield independent components. In global register allocation, an example of an unstructured global compiler optimization, it is generally not the case that the partial solution for an inner loop can be incorporated into the solution of the enclosing loop without re-computation. The lack of easily detectable data locality greatly influences possible parallelizations of such a problem. The challenge for such optimization is to find a model that permits us to incorporate enough knowledge about the specific problem instance such that data partitioning points can be found, though usually not at locations that coincide with loop or conditional entries and exits.

1.5. Two representative optimizations

To assess how much parallel frameworks for global compiler optimizations depend on program structure, we develop parallel frameworks for both a structured and an unstructured global compiler optimization. We concentrate on interval analysis of global data flow problems and on global register allocation as representatives for both classes. This choice was guided by the generality of both optimizations; both are machine independent and important optimizations used in a number of optimizing compilers.

1.5.1. Parallel interval analysis of data flow equations

Interval analysis provides a structured framework for *all* global data flow problems. The purpose of global data flow analysis is to collect information about the flow of data for every basic block in a program. Global data flow analysis is prerequisite for all global compiler optimizations. Some optimizations like for instance dead code removal or common subexpression elimination change the data flow information and therefore global data flow analysis must be carried out several times during global optimizations. A parallelization of interval analysis can be expanded to a parallelization of all flow problems.

Data parallelism based on explicit program structure

In global data flow analysis, locality of data is given explicitly by the loop structure of a program. Interval analysis is a method in which the data flow information is gathered on a per-loop basis, starting with the innermost loops. The data flow information for an inner loop need not be re-computed for the data flow information of the enclosing loop so in other words, interval analysis does not require backtracking. The loop structure of the program is the basis for the data partitioning for our parallelization of interval analysis. Because data locality is given explicitly by the loop structure, our parallelization of interval analysis is straightforward. We evaluate our approach by a parallel implementation and present our measurements of the parallel speedup over a standard sequential implementation.

1.5.2. Data partitioning in global register allocation

Global register allocation is the problem of mapping variables and temporary variables created by compiler optimizations to registers efficiently. Because access times to registers are considerably lower than access times to memory, it is desirable to keep as many variables as possible in registers during program execution. Registers are a scarce resource on every computer architecture, therefore the number of registers is usually no sufficient to hold all variables of an input program. Optimal global register alloction is mathematically equivalent to finding an optimal coloring for an arbitrary graph and therefore NP complete, but there are certain types of graphs for which am optimal coloring can be found in polynomial time. *Interval graphs* can be colored optimally in linear time and are particularly important in the context of parallel global register allocation via graph coloring for two reasons. First, many register conflict graphs contain portions that are interval graphs, and second, it is easy to partition interval graphs into clique connected components that can be colored independently. An overall coloring of clique connected components can be obtained by renaming only; re-computing partial results is not necessary [Tarjan 85].

While for interval analysis data locality is given in a natural way by the loop structure of the input program, finding a data partitioning for global register allocation is not straightforward. The focus of the thesis research in parallel global register is the development of a structured model for data partitioning based on clique separators in the register conflict graph. Knowledge about program structure is used to detect *interval* portions of the register conflict graph in which clique separators can be found systematically. Non-interval portions of the register conflict graph are manipulated such that the derived graph is an interval graph. Once the interval portions of a register conflict graph are known, partitioning the conflict graph via clique separators and the parallelization are straightforward.

The evaluation of our method to detect clique separators in register conflict graphs focusses on determining whether the model is powerful enough to detect enough clique separators in the register conflict graphs of real programs.

1.6. Approaches to parallel compilation

The coarsest level of parallelism in compilation is *parallel system building*. Prerequisite for parallel system building is separate compilation. A system that consists of several modules is composed to a runfile at link time - each moule can be compiled in parallel before linking. Optimistic make [BubZwaen 92], a system that distributes the compilation of individual modules applies this level of parallelism.

The compiler used in parallel system building is still sequential - the next level of parallelism in compilation is to turn a sequential compiler into a parallel program. The coarsest units of parallelism are procedures and functions. If procedures and functions are to be compiled independently, inter procedural optimizations have to be curtailed. This type of parallelization has been studied before [Frankel 83, GrossZobel 89]; it requires the compiler driver to dispatch the (sequential) compilation of each function or procedure. This approach is very successful when inter procedural optimizations are curtailed because then there are no dependencies between individual parallel tasks. Note that the core of the compiler still runs sequentially.

The next level of parallelism is achieved by compiling units that are smaller than procedures in parallel. Many research efforts focus on parallel parsing [Boehm 87, Klein 90, Fischer 75], the compilation phase that is formally best understood. Both the parallel compiler developed at the University of Toronto [Seshadri et al 88] and the parallel compiler described in [Vandevoorde 88] are examples of parallel one pass compilers. The input program is divided into components during parsing, and each component is processed independently through code generation. This method works well in practice for non-optimizing compilers or compilers that apply local optimizations only. For global optimizations, interactions between any parts of a program must be considered. Partitioning the input program at parsing time makes it hard to predict the communication traffic between the components during the global optimization phase, so potential parallel speedup is offset by interactions between the components during global optimizations.

To date, the parallelization of global optimizations has received little attention. Some efforts have conentrated on the parallelization of global data flow analysis [LeeMarloweRyder 91, GuptaPollockSoffa 90], and approaches to perform global register allocation hierarchically or incrementally are indirectly related to detecting data parallelism in the problem [GupSofSte 89, CallahanKoblenz 91].

What distinguishes this thesis from other approaches to the parallelization of global compiler optimizations is the development of models that relate program structure to data parallelism. Main contribution of this thesis is the development of a structured model for detecting data parallelism in global register allocation. The novelty of the approach taken in the thesis is that program structure is used systematically rather than heuristically. The analysis of boundary conditions between components that are processed independently ensures that partial results are combined without compromising optimality of the overall result.

1.6.1. Organization of the thesis

The remainder of this thesis is organized as follows. Our model of compilation and basic definitions are given in Chapter 2. The parallelization of interval analysis and the evaluation of a parallel implementation are presented in Chapter 3. The bulk of the thesis is devoted to the model for structured global register allocation. Terminology and basic definitions of the model are given in Chapter 4. In Chapter 5 we demonstrate techniques that allow us to detect which portions of a register conflict graph are interval graphs. Manipulations that can change a non-interval register conflict graph into an interval register conflict graph are introduced in Chapter 6. The evaluation of the model for structured global register allocation is given in Chapter 7, and we conclude by discussing the contributions in Chapter 8.

Chapter 2
Background

Goal of this thesis is to investigate methods to parallelize global compiler optimizations. We restrict our input programs to well structured programs built from a set of compound programming constructs. In this chapter we introduce our model of compilation, and then give definitions of compound programming constructs and well structured programs. The definitions given in this chapter will be used throughout the remainder of the thesis.

2.1. A model for compilation

A compiler consists of one or several phases, some of which are formally better understood than others. Before going into detail about the problems of parallel optimization, we present an "abstract" model of compilation. This abstract model contains all important compilation phases and illustrates the dependencies between the phases. A compiler implementation consists of a combination of the phases in the abstract model. An enumeration of the phases in our compilation model is listed below.

Parsing and semantic checking
Derive the syntax tree from the input and check for syntactic and semantic errors in the program.

Intermediate code generation
Derive the intermediate language representation of the program from the syntax tree. Partition the program into basic blocks and construct the program flow graph.

Local data flow analysis
Perform data flow analysis within the basic blocks of the program. Local data flow analysis is necessary to perform local optimizations.

Local optimizations Perform optimizations within basic blocks.

Global data flow analysis
Compute information about the flow of data in the entire program, i.e. *across* basic block boundaries.

Global optimizations
Optimize the code across basic blocks, that is, consider the flow of control of the entire program. Global optimizations include inter procedural optimizations.

Code generation Generate code from the (possibly optimized) intermediate language representation and convert the intermediate language representation into assembly code.

Assembly Produce machine code from assembly code; perform peephole optimizations.

Figure 2-1 gives a view of the compilation model. In this figure, the boxes represent compilation phases, and arcs between the boxes depict data dependencies between the phases. Every compilation starts with

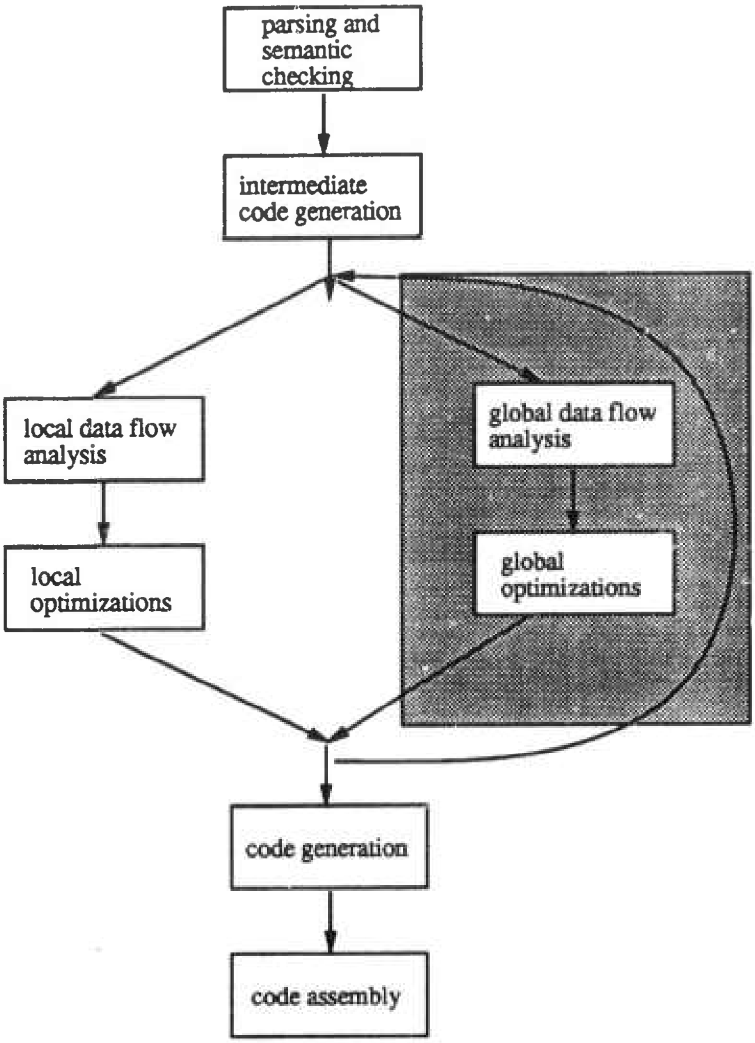


Figure 2-1: The compilation model

parsing and intermediate language generation. Non-optimizing compilers enter code generation directly after parsing and semantic checking. Data flow analysis and optimization occur after intermediate code generation and before code generation. Optimizing compilers can be classified in *one pass* compilers and *multiple pass* compilers. One pass compilers perform each stage of the compilation exactly once while multiple pass compilers loop through the optimization phases several times, each time performing more optimizations.

The shaded are of Figure 2-1 depicts the parts of the compilation process that are the research focus of the thesis.

2.2. Input model

In out compilation model, global optimizations succeed the intermediate code generation phase, so the input program has been divided into basic blocks and the program flow graph has been constructed. In other words, the input to the global optimization phase are flow graphs constructed from the input program. Throughout this thesis, the smallest unit of parallelism is a *basic block*, defined as follows:

> **Definition 1:** *(Basic block)* A basic block consists of a nonempty set of instructions $i_1,...,i_n$ such that no instruction $i_j \in \{i_1,...,i_{n-1}\}$ is a jump instruction. The last instuction i_n can be but need not be a jump instruction. The first instruction i_1 can but need not be the target of a jump instruction.

Note that our definition of basic blocks differs from the most commonly used definition of basic blocks where the first instruction is always the target of a jump instruction and the last instruction is always a jump instruction.

Before we give our definition of a well structured flow graph, we define individual programming constructs that occur in well structured flow graphs.

Note that every basic block consists of a straight line code sequence, and that basic blocks can consist of as few as one instruction. Because we do not require that the last instruction of a basic block be a jump instruction, we define straight line code in terms of basic block sequences.

> **Definition 2:** *(Straight line code)* Straight line code is a directed graph consisting of basic blocks $b_1,...,b_n$ such that all $b_i \in \{b_1,...,b_n\}$ are connected and each $b_i \in \{b_1,...,b_n\}$ has exactly one incoming and one outgoing edge.

> **Definition 3:** *(Loop)* A *loop* is a directed graph consisting of a loop head *h*, a loop exit *e* and a loop body. The loop body can consist of straight line code, a conditional branch or a loop. A cycle in the loop contains *h* iff it contains *e*.

Figure 2-2 shows some examples of loops. The leftmost construct is a nested loop, the loop in the middle consists of a single basic block, and the rightmost loop consists of a conditional statement.

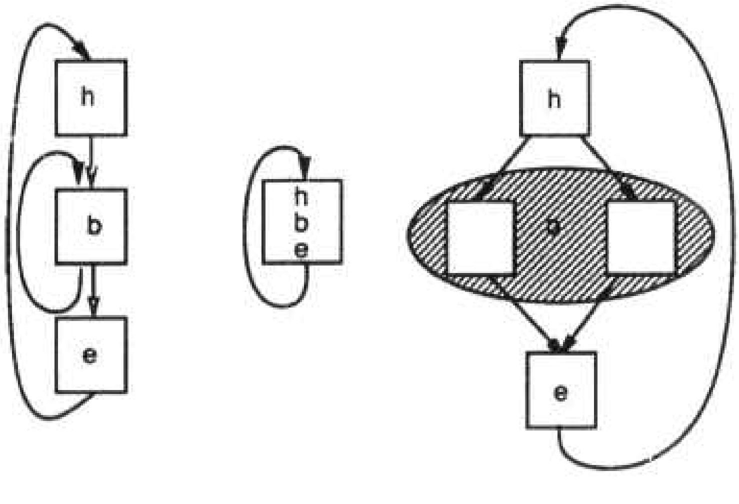


Figure 2-2: Examples of loop constructs

> **Definition 4:** *(Conditional branch)* A *conditional branch* is a directed graph consisting of a *split* node, a *join* node and a set of *branch clauses*. A branch clause consists of either straight line

code, a loop or a branch. No computations are performed in either the split or the join node. Each branch clause contains a node *s* such that there is an edge from the split node to *s*. Each branch clause contains a node *e* such that there is an edge from *e* to the join node.

The split node and the join node are "imaginatory" basic blocks in which no computation takes place. Given a node *n* in a flow graph that has more than one outgoing edges that are not backedges, we insert a new node *s* and an edge from *n* to *s*; all forward edges originating at *n* originate from the newly introduced node *s*. Given a node *m* in a flow graph that has more than one incoming edges that are not backedges, we insert the join node *j*, re-direct all forward edges that go into *m* into *j* and insert an edge from *j* to *m*. Because no computations are performed in *s* and *j*, the data flow information does not change for any basic block in the original flow graph. We will use the split and join node to distinguish between live ranges of variables that occur in conditionals in our model for global register allocation.

Figure 2-3 shows a conditional branch with split node *B0'*, join node *B8'* and two branch clauses. The first branch clause consists of a conditional branch consisting of *B1*, *B2*, *B3* and *B4* with inserted split and join node *B1'* and *B4'*, the second branch clause consists of a loop, formed by *B5*, *B6* and *B7*. The split and join nodes have been inserted into the original flow graph; this is indicated by representing them as circles.

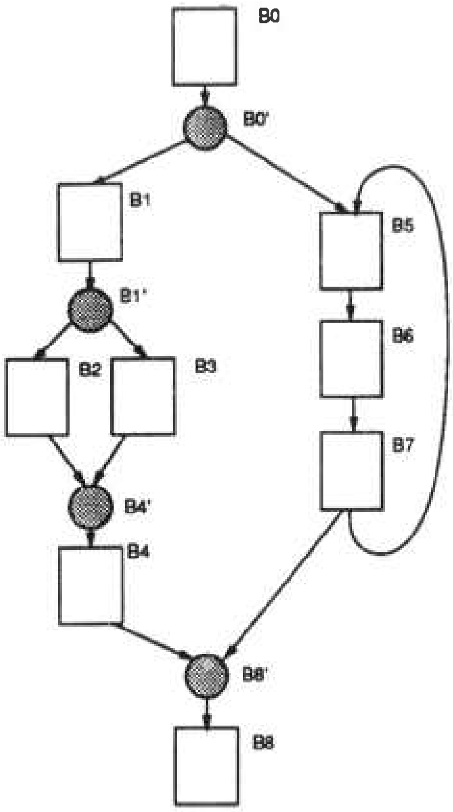


Figure 2-3: A conditional branch

Note that our definition of conditional branches requires one unique split node *S* and one unique join node *J* per conditional. The pair <*S*,*J*> uniquely identifies a conditional. Note that the number of branch clauses is unlimited. Thus, a case statement is just an example of a conditional branch.

We now give the formal definition of a well structured flow graph.

Definition 5: *(Well structured flow graph)* A well structured flow graph is a directed graph consisting of basic blocks and a set of edges, such that

1. There exists one unique basic block called the *program entry* which dominates all other basic blocks in the flow graph.
2. There exists one unique basic block called the *program exit* which has no outgoing edges
3. The graph derived from the flow graph by removing the program entry and the program exit and all edges incident to those nodes consists of a sequence of pieces of straight line code, loops and conditional branches.

2.3. Chapter summary

We have introduced our model of compilation and defined our input model for global optimization. We restrict our input to well structured flow graphs that consist of straight line code, loops and conditionals. Throughout the thesis, we refer to the definitions given in this chapter whenever one of those terms is used.

Chapter 3
Parallel Interval Analysis

In this chapter, we describe the development of a parallel framework for global data flow analysis. Global data flow analysis is a prerequisite for all global compiler optimizations. The purpose of global data flow analysis is to collect information about the flow of data for every basic block in a program flow graph. Given a program flow graph with n basic blocks, the time complexity of solving a global data flow problem is $O(n^2)$. From a theoretical standpoint this is small; the reason why global data flow analysis can account for a considerable portion of the overall compilation time is that global data flow problems must be solved many times in the course of global optimization.

Interval analysis is a framework for global data flow analysis in which the basic blocks of a program are processed in a particular order. The reason why interval analysis is a good candidate for parallelization is that the data processed during interval analysis can be partitioned into independent pieces in a natural way. The data flow information for non-nested loops can be computed independently and embedded into an overall result without backtracking. This leads to a straightforward parallelization of interval analysis: the loop structure of the program dictates the partitioning of the data for the parallelization. A number of global optimizations that operate on loops of a program fit into the same parallel framework; one example is loop vectorization. To assess the effectiveness of this simple parallelization we implemented parallel interval analysis for the solution of a data flow equation and measured the parallel speedup for a set of benchmark functions.

After a brief introduction of interval analysis, we describe our parallel framework for interval analysis and its implementation. We discuss approximations to the optimal parallel speedup and conclude the chapter with the description of our implementation and the presentation of measurements of the observed speedup for a benchmark of functions.

3.1. Introduction to global data flow analysis

The purpose of global data flow analysis is to compute global information about variables and expressions in a program. Global data flow problems are formally well understood, and there exist a number of different frameworks for the computation of data flow information. The most common techniques for global data flow analysis are discussed in [Kennedy 81]. In this section we give a brief introduction to data flow analysis and interval analysis. We illustrate our introduction with an example of a data flow problem.

Data flow analysis problems can be divided into two classes:

Class 1	Given a basic block b in the flow graph, what can happen *before* control reaches b, that is, what definitions can affect computations at b.
Class 2	Given a basic block b in the program, what can happen *after* control leaves b, that is, what uses of expressions can be affected by computations at b.

Class 1 problems are known as *forward flow* problems, class 2 problems are called *backward flow* problems. An example of a class 1 problem is *availability analysis*, a typical class 2 problem is *liveness analysis*.

Global data flow problems can be formulated as a set of *data flow equations*. The core of global data flow analysis is to find solutions for those equations.

The simplest approach to data flow analysis is to iterate through the nodes of the flow graph applying the appropriate equations until no changes take place. If the number of nodes in the flow graph is n, the iterative algorithm requires $O(n^2)$ steps for the entire computation [Aho 84].

Interval analysis is a method to solve data flow equations that takes advantage of the locality of the flow information in loops. Interval analysis consists of two passes. In Pass 1 of interval analysis, local information for each basic block is determined in a form suitable for solving the equations. Once the local properties of all basic blocks are known, the second pass determines the interactions with other basic blocks. After Pass 2 is finished, we know how each basic block is affected by the instructions of every other basic block.

3.2. Interval analysis: background

Interval analysis operates on regions or intervals of a flow graph in a specific order. Intervals capture loops in a flow graph, more formally:

Definition 1: *(Interval)* Given a flow graph G with basic blocks B, an interval in G is defined to be a set of basic blocks $I \subseteq B$ with the following properties:

- There is a node $h \in I$, called the *head* of I, which is contained in every path from a block outside I to a block within I. In other words, I is a single entry region.
- I is connected.
- $I - \{h\}$ is cycle-free; i.e., all cycles within I must contain h.

The order in which nodes are added to an interval I is called *interval order*. The interval order is significant, in that if nodes of I are processed in interval order, a particular node will be treated only after all its predecessors have been processed.

Intervals correspond to loops in the program. The intervals of the original flow graph represent the innermost loops. The graph derived from a flow graph G by replacing the set of nodes that form an interval by *one* node representing the interval is called a *derived flow graph* of G. More formally:

Definition 2: *(Derived flow graph)* Given a flow graph G with initial node n_0, the derived flow graph $I(G)$ is the following graph:

1. The nodes of $I(G)$ are the intervals found by interval partition of G.
2. If J and K are two intervals, there is an edge from J to K in $I(G)$ if and only if there exist nodes $n_j \in J$ and $n_k \in K$ such that n_k is a successor of n_j in G.
3. The initial node of $I(G)$ is the interval including n_0.

Given a node h in a flow graph G, the interval I for h is built by adding all basic blocks of G not in I whose predecessors are already in the interval I; the detailed algorithm is given in Figure 3-1.

```
Input: The specified head h.
Output: max_interval[h]
begin
  I := {h};
  while ∃x∈ (S[I]–I)  such that P[x] ⊂ I
     do
       I := {h};
     od;
  max_interval(h) := I;
end
```

Figure 3-1: Algorithm to construct the maximum interval for a given head

Figure 3-2 shows a flow graph - the shaded areas depict the initial partition into intervals. Note how each individual loop is captured in one interval.

Interval partition is repeated until the derived graph consists of only one single node, called the *limit flow graph*. We call the sequence of graphs derived by repeated interval partition that starts with a flow graph G the *derived sequence* of G.

Irreducible flow graphs

Flow graphs, for which a limit flow graph that consists of only one node does not exist, are called *irreducible*. Flow graphs, whose limit flow graph consists of only one single node, are called *reducible*. Flow graphs of well structured programs are always reducible, and there are techniques for changing irreducible flow graphs into reducible flow graphs [Aho 84]. For the purpose of this thesis we assume well structured programs, hence we do not address the irreducibility of flow graphs in the remainder of this chapter.

Definition 3: *(Derived sequence of a flow graph)*: Given a flow graph G, the sequence of graphs $(G_0,G_1,...,G_m)$ is called the *derived sequence* for G if

- $G = G_0$
- G_j is derived by interval partition from G_{j-1} $\forall j \in \{1,...,m\}$
- G_m is the limit flow graph for G_0

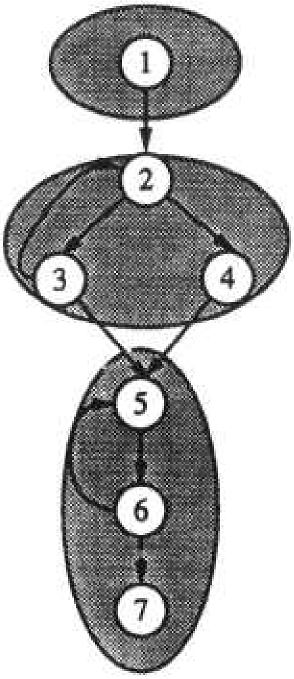


Figure 3-2: A flow graph and its initial interval partition

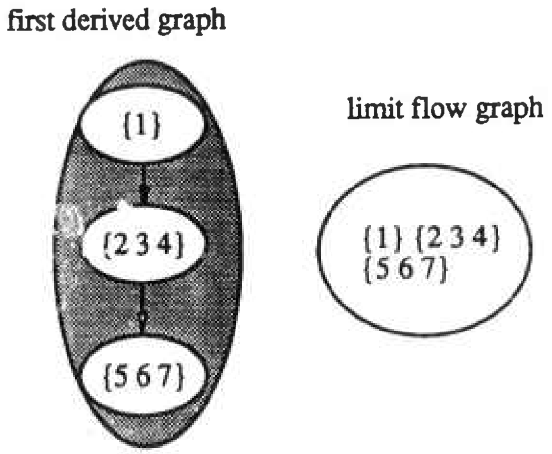


Figure 3-3: Sequence of derived graphs

Figure 3-3 shows the derived sequence for the flow graph depicted in Figure 3-2. The shaded regions of the flow graph graph shown in Figure 3-2 depict the intervals after the first interval partition, which are the nodes of the first derived flow graph. The second generation of interval partition yields the limit flow graph.

Interval partition gives rise to a two-pass algorithm for data flow analysis. In the following, the method is discussed as it applies to an availability system.

3.2.1. Availability analysis

An expression e is said to be *available* at a basic block b in the program if it has been computed before control reaches b, and none of its operands have been destroyed between its computation and b. *Local availability* means availability within one basic block, *global availability* means availability within the entire flow graph.

An expression e is *redundant* at point p if e is available on *all* paths that lead to p. Availability is one of the key problems in program optimization because it allows to identify redundant expressions which may be eliminated from the program.

We now introduce an availability system that was developed by Morel and Renvoise [MorelRenvoise 81]. We first explain the local properties of basic blocks which are needed to solve the availability system.

Up-transparency *UTRANSP*

A block is said to be *up-transparent* for an expression if the block does not contain any modification of the operands of the expression, or if the first modification of an operand of the expression occurring in the block is preceded by a computation of the expression.

Down-transparency *DTRANSP*

A block is said to be *down-transparent* for an expression if the block does not contain any modification of the operands of the expression, or if the last modification of an operand of the expression occurring in the block is followed by a computation of the expression.

Local availability *COMP*

An expression e is said to be *locally available* in a block i if there is at least one computation of the expression in the block i, and if the instructions appearing in the block after the last computation of the expression do not modify e's operands.

We will use $AVIN_i$ to denote global availability upon *entry* of block i and $AVOUT_i$ to denote global availability upon *exit* from block i. Further, boolean conjunctions are denoted $\bullet$ and $\prod$, disjunctions + and $\sum$ respectively. In the following, let B denote the set of all basic blocks.

In this notation, an expression is available on entry to a block if it is available on exit from each predecessor of the block. An expression is available on exit from a block if it is locally available, or if it is available on entry to the block and down-transparent in this block. This leads to the following set of equations for global availability:

$$AVOUT_i = COMP_i + DTRANSP_i \bullet AVIN_i$$
$$AVIN_i = \prod_{j \in Pred(i)} AVOUT_j, i \in B$$

3.2.2. Pass 1 of interval analysis

During the first pass, local quantities *COMP*, *DTRANSP* and *UTRANSP* are computed for larger and larger regions of the program. The algorithm *I1* depicted in Figure 3-4 computes *COMP*, *DTRANSP* and *UTRANSP* for an interval from their values for blocks in the interval. If $G_0, G_1, \ldots, G_m$ is the derived sequence of flow graph $G = G_0$, Pass 1 consists of applying algorithm *I1* to each interval in G_0, then to each

```
Input:            an interval I with head h
                  COMP_x, DTRANSP_x and UTRANSP_x for each x ∈ I
Output:           COMP_I, DTRANSP_I and UTRANSP_I
Miscellaneous:    Succ(x) are the successors of x
Method:

        begin

                  $COMP_I = COMP_h$;
                  $DTRANSP_I = DTRANSP_h$;
                  $UTRANSP_I = UTRANSP_h$;

                  for all $x \in I - \{h\}$ do
                    for all $y \in Succ(x)$ do

                      $COMP_I = (COMP_I \bullet UTRANSP_x) + COMP_y$
                    od;
                  od;
                  for all $x \in I - \{h\}$ do
                    $DTRANSP_I =$
                        $(DTRANSP_I + UTRANSP_I) \bullet DTRANSP_x$
                    $UTRANSP_I =$
                        $(UTRANSP_I \bullet DTRANSP_I) + UTRANSP_x$
                  od;
        end
```

Figure 3-4: Algorithm *I1*

interval in G_1, and so on. When G_{m-1} is reached, *COMP*, *DTRANSP* and *UTRANSP* will have been computed for each node in the derived sequence of graphs.

3.2.3. Pass 2 of interval analysis

```
Input:            an interval I with head h
Output:           AVIN and AVOUT for all members of I
Miscellaneous:    Pred(x) are predecessors of x
Method:
        begin
                  for all $x \in I$ in interval order, starting with the head h do

                           $AVIN_x = \prod_{j \in Pred(x)} AVOUT_j$;
                        $AVOUT_x = COMP_x + DTRANSP_x \bullet AVIN_x$;

                  od;
        end
```

Figure 3-5: Algorithm *I2*

During the second pass, *AVIN* and *AVOUT* are computed for smaller and smaller regions of the program.

If x^* denotes the single node in G_m, Pass 2 begins with the assignment $AVIN_{x^*} = TRUE$. The remaindur of the pass consists of repeated application of algorithm *I2* shown in Figure 3-5, which computes *AVIN* and *AVOUT* for each node in an interval *I*, given correct *AVIN* and *AVOUT* sets for the entry to *I* and to each successor *J* of *I*. *I2* is applied first to the interval of G_m, then to the intervals of G_{m-1} and so on until *AVIN* and *AVOUT* have been computed for every node in the original graph *G*.

3.3. Parallel interval analysis

The key idea in our parallelization of interval analysis is that disjoint intervals contain disjoint sets of basic blocks and therefore may be treated independently. Another important observation is that an interval I_j in the *i-th* derived flow graph depends only on a *subset S* of the basic blocks. Thus, not only the nodes of the *i-th* derived graph G_i but all nodes in G_0, G_1, ..., G_i independent of *S* may be processed in parallel with interval I_j.

Recall that during Pass 1 algorithm *I1* is applied first to the nodes of the original flow graph, then to the nodes of the first derived flow graph etc. Thus, the level of parallelism decreases as we progress through the sequence of derived flow graphs. In Pass 2, algorithm *I2* is first applied to the limit flow graph, then to the previous derived flow graph etc. Therefore the level of parallelism in Pass 2 increases as we progress through the (reversed) sequence of derived flow graphs.

The order in which the nodes of the derived sequence are processed corresponds to a postorder traversal of a tree composed of the nodes in the graphs of the derived sequence, called *complete interval tree*.

Definition 4: *(Complete interval tree)* Given a program flow graph *G* and its derived sequence $\{G_0, G_1, ..., G_m\}$, the complete interval tree of *G* is a tree with the following properties:

- The leaves of the tree are basic blocks.
- Every node in the tree corresponds to an interval in a graph in the derived sequence $\{G_0, G_1, ..., G_m\}$.
- There is an edge from node *J* to node *K* iff the interval *J* of derived flow graph G_m contains the interval *K* of the previous derived flow graph G_{m-1}.
- The root of the tree is G_m, the limit flow graph.

Eliminating redundant nodes from the tree

Figure 3-7 shows a complete interval tree in which each node *n* has an associated weight that equals the number of leaf nodes in the subtree rooted by *n*. The weight of a node corresponding to an interval is therefore the number of basic blocks in that interval. The intervals of the *i+1*st derived flow graph are composed of the nodes of the *i*th derived flow graph.

The sequence of the derived flow graphs reflects the loop structure of a program: the first derived flow graph reflects the innermost loops, the second level loops that enclose the innermost loops etc. So the weight of intervals containing the basic blocks that form loops increases with the index of the derived flow graph.

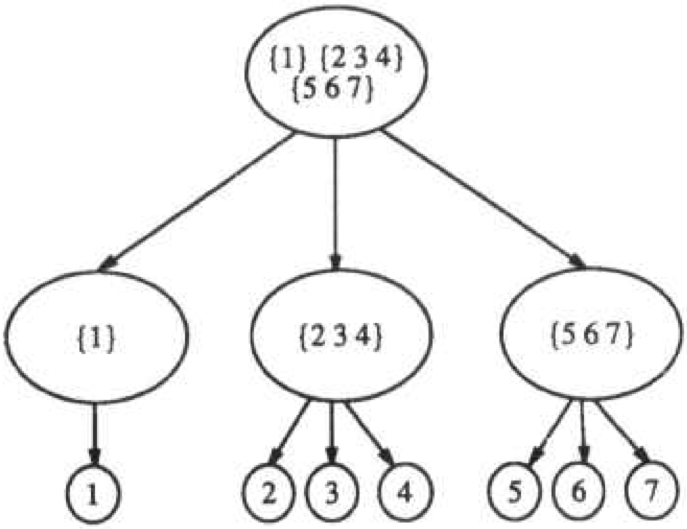


Figure 3-6: Example complete interval tree

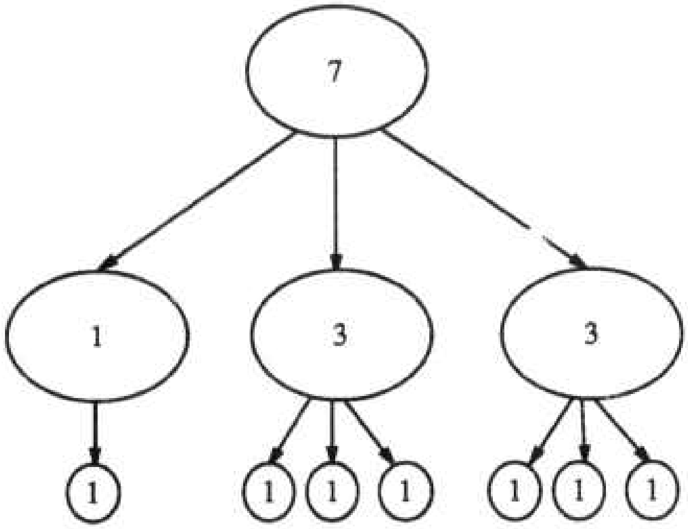


Figure 3-7: Weighted interval tree

Looking at our example in Figure 3-7, the associated weight of nodes in the leftmost branch of the tree does not change until the root. Thus, in each iteration of building the derived flow graphs the intervals represented by the leftmost nodes in the tree contain the same one basic block. Therefore, the data flow information for these nodes does not change from one iteration to the next. This suggests the following ***elimination rule***: Nodes of the *i+1*st derived flow graph that have only one child (i.e. whose weight is no larger than the child's weight) need not be reconsidered during the flow analysis pass and can be eliminated from the complete interval tree. Figure 3-8 depicts the same complete interval tree after all nodes that do not change the data flow analysis information have been eliminated. Eliminating redundant nodes from the complete interval tree means to avoid unnecessary overhead of scheduling redundant nodes during processing.

The complete interval tree is the basis of our parallelization of interval analysis. Algorithm *I1* is applied

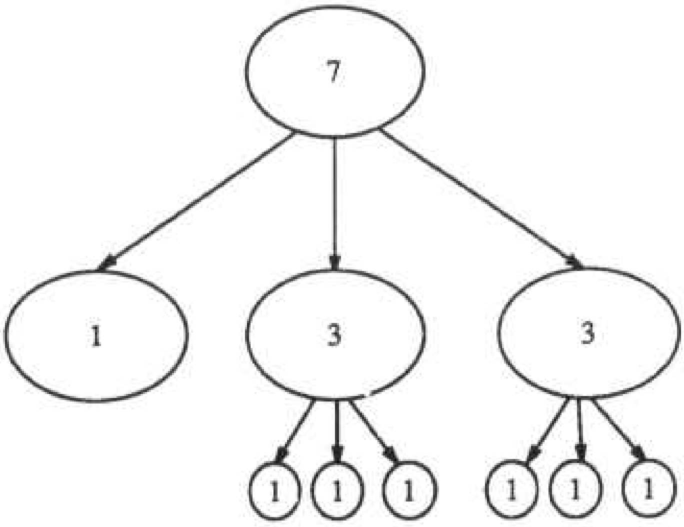


Figure 3-8: Interval tree after elimination

to the complete interval tree bottom up, starting at the leaves of the tree, computing the *COMP*, *DTRANSP* and *UTRANSP* sets of the program for bigger and bigger regions of the flow graph. Upon completion of algorithm *I1*, *I2* computes the *AVIN* and the *AVOUT* sets of variables for smaller and smaller regions of the program, starting at the root of the complete interval tree (which denotes the entire program) and ending at the leaves of the tree.

The order in which algorithms *I1* and *I2* have to be applied to the nodes of the complete interval tree is only restricted by the parent/child relations of the nodes. In Pass 1, a node *J* can only be processed after all its successors have been processed. In Pass 2, a node *J* can only be processed after all its parent nodes have been processed. Therefore, all nodes that are not ancestors or descendants of each other can be processed independently. The execution of nodes that are in ancestor relationship must be synchronized.

3.4. A model to approximate the amount of parallelism in parallel interval analysis

Assuming that the processing times for each individual node are known, the minimal processing time for a given complete interval tree is bounded by the longest path from the root to a leaf node, where the length of a path is the sum of the processing times of its nodes, more formally:

Definition 5: *(Length of a path in a complete interval tree)* Given a complete interval tree T with root R, nodes N and processing time $t(n)$ for all nodes $n \in N$, let p be a path from a leaf node to the root R consisting of $N' \subseteq N$. Then, the length of p $L(p)$ is defined as

$$L(p) = \sum_{n \in N'} t(n)$$

The formal definition of the minimal processing time is then:

Definition 6: *(Minimal processing time for a complete interval tree)* Given a complete interval tree T with root R, let P be the set of all paths from the leaf nodes of T to the root R. The minimal processing time for T, $t_{min}(T)$ is the length of the longest path among P:

$$t_{min}(T) = max\{L(p) | p \in P\}$$

The most efficient parallelization processes the tree in minimal time and utilizes all processing elements

100% of the time. A tree structure as basis for parallelism does not permit 100% processor utilization if the number of processing elements exceeds 1. The reason is that the root node can not be processed in parallel with any other node, so all processors except one are idle when the root node is ready for processing.

The number of processors that can be used efficiently for processing a complete interval tree in minimal time depends on the shape of the complete interval tree. Figure 3-9 shows complete interval trees of different shapes.

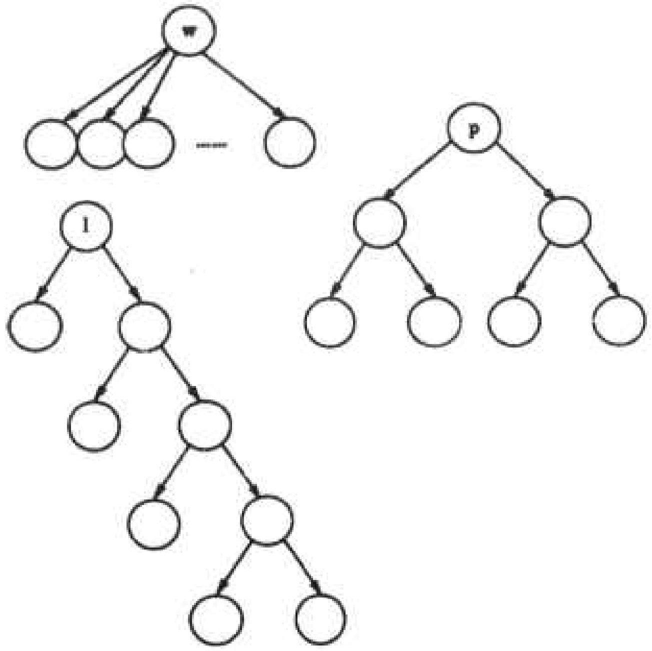


Figure 3-9: Differently shaped complete interval trees

The three examples in Figure 3-9 represent the spectrum of shapes that complete interval trees can have. The tree with the root labeled *w* is wide and flat. Recall that the leaf nodes of complete interval trees are the basic blocks of the program. A program that results in a complete interval tree of that form contains no loops. A program that consists of one deep nest of loops results in a complete interval tree that looks like the example labeled *l* in Figure 3-9: the innermost loop is found at the bottom of the tree, basic blocks that are parts of the enclosing loops are accumulated on the way from the bottom to the root. The tree labeled *p* is a perfectly balanced binary tree, resulting from several independent loop nests in the program.

Intuitively trees that are wide and bushy can keep a large number of processors busy; trees that are long and thin do not permit much parallelism. It is easy to determine the minimal processing time for a given complete interval tree when the processing time for each node is known. Given an unlimited resource of processors, every complete interval tree can be processed in minimal time. Given only a fixed number of processors, it is hard to determine a priori whether a given tree *can* be processed in minimal time. The following paragraphs address the problem of establishing a lower bound for the number of processors *needed* to process a complete interval tree in minimal time. A formal definition of this bound is given below:

Definition 7: *(Parallel efficiency of a complete interval tree)* Given a complete interval tree *T* with minimal processing time t_{min}, the parallel efficiency of T is the minimal number of processors needed to process T in t_{min}.

The parallel efficiency of a complete interval tree T is a measure for the "amount" of parallelism in T. If the parallel efficiency for a complete interval tree T is known, the speedup observed in a parallel implementation can be analyzed appropriately. If for example the parallel efficiency of a complete interval tree is p, and $p'>p$ processors operate in parallel, there exists a schedule for executing the nodes in the complete interval tree with p processors such that the speedup observed with p' processors does not exceed the speedup observed with p processors. The reason is that only p processors can be used efficiently, any additional processor is bound to be idle a lot of the time.

We start the discussion by showing some properties of complete interval trees that allow us to establish an upper bound on the parallel efficiency of a complete interval tree.

3.4.1. Properties of complete interval trees

Basic blocks and nodes in the complete interval tree are the smallest units processed in parallel. Before we reason about the parallelism in the tree, we state some properties of complete interval trees for reducible flow graphs. These properties permit to compute certain bounds for the width and the height of such trees.

Lemma 8: Given a complete interval tree T in which all redundant nodes have been eliminated by the elimination rule, every node in T that is not a leaf node has at least 2 children.

Proof: Each node in the complete interval tree corresponds to either a basic block in the flow graph of a program or to an interval in the flow graph. Basic blocks form leaf nodes in the tree and have no children. By the elimination rule, nodes that correspond to intervals that consist of only one node do not occur in T. Therefore, only intervals that consist of at least two nodes are represented. By construction of complete interval trees, such nodes must have at least two children.

Lemma 9: Given a complete interval tree T of a program flow graph G with n basic blocks, the height of T is at most n.

Proof: Induction on n, the number of basic blocks. $n=1$: A flow graph consisting of one basic block is equivalent to the corresponding limit flow graph, therefore the complete interval tree consists only of one node and has height 1. n->$n+1$: Under the assumption that T's height is at most n let G' be the flow graph derived from G by adding a basic block b at an arbitrary location. The corresponding interval tree T' can differ from T as follows:

1. b is part of an inner loop, in which case the interval in G that corresponds to this loop will consist of one more basic block - the height of T' equals the height of T

2. b starts a new interval and therefore T' can have at most one more node - therefore the height of T' can be at most the height of $T+1 = n+1$

Lemma 10: Given a complete interval tree T of a program flow graph G with n basic blocks, the number of nodes in T is at most $2n-1$.

Proof: T is composed of basic blocks and nodes of the derived flow graphs. Each node in a derived flow graph has at least two children, otherwise it would be eliminated by the elimination rule. In the worst case there can be at most one node in any derived flow graph per basic block - again because of the elimination rule. Further, there must be at least one interval that contains two basic blocks. If the number of basic blocks is n, the total number of interval nodes is bounded by $n-1$, hence the total number of nodes in T is bounded by $2n-1$.

During the processing of a complete interval tree T, only a subset of nodes are executable at any given time. In the beginning, only leaf nodes are executable; an inner node n in the complete interval tree becomes executable once all the nodes in the subtree rooted at n have been processed. Before we compute an upper bound on the number of nodes in a complete interval tree that can be executable simultaneously, we give a formal definition for that quantity.

Definition 11: *(PARSET of a tree)* Given a complete interval tree T with minimal processing time t_{min}, let $t < t_{min}$. *PARSET(t)* is the set of nodes in T that are executable at time t.

Note that the size of the *PARSET* of a complete interval tree is independent of the number of processors. In the next lemma, we state that the *PARSET* of a complete interval tree can not exceed a certain number.

Lemma 12: Given a complete interval tree T of a program flow graph with n basic blocks with minimal processing time t_{min}, $PARSET(T) \leq n$.

Proof: *Case 1*: *PARSET* consists exclusively of basic blocks. In that case $|PARSET(t)|$ is certainly at most n.

Case 2: *PARSET* consists of both interval nodes and basic blocks. Let K be an arbitrary node in *PARSET(t)*. Key observation in the proof is that while K is processed, *PARSET(t)* can not contain any node in K's subtree or any node on the path from K to the root. Let *PARSET(t)* contain k_1 basic blocks, k_2 nodes of the first derived flow graph, ..., k_l nodes of the last derived flow graph. By Lemmas 9 and 10, the number of nodes in the subtree of a node of the *j-th* derived flow graph is at least $2i$ and the number of nodes along the path from that node is $n-j$. Thus the number of nodes in *PARSET(t)* contains at most $(2n-1) - \sum_{i=0}^{i=l} (k_i \times 2i + n - j) \leq n$ q.e.d.

As a result of Lemma 12, our parallelization can keep at most n processors busy simultaneously if n is the number of leaf nodes in the complete interval tree. There are cases in which the complete interval tree can be processed in minimal time with fewer than n processors, where n is the number of leaf nodes of the complete interval tree. This is illustrated in Figure 3-10. Under the simplifying assumption that every node requires the same processing time, the longest path from the root to a leaf node consists of nodes {1,3,5,7} and therefore the minimal processing time for this tree is 4. The number of leaf nodes is 4, but the tree can be processed in minimal time with just 2 processors; this is achieved by processing nodes {6,7}, then {4,5}, then {2,3} and finally the root, {1}. The parallel efficiency of that tree is 2; we will show in the following sections that computing the parallel efficiency for a given interval tree is NP complete; we conclude our theoretical analysis by computing upper and lower bounds for the parallel efficiency of complete interval trees.

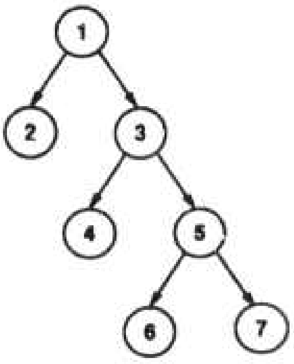


Figure 3-10: Minimal time with fewer processors than leaf nodes

3.4.2. Bounds on the parallel efficiency of complete interval trees

To be able to compute the parallel efficiency of a given interval tree, we must know the processing times for all nodes. Usually, the processing times for different nodes in a realistic complete interval tree vary considerably. In the following, we assume that for every node in the complete interval tree the computation time is known.

The next lemma states that determining the parallel efficiency of an arbitrary complete interval tree is NP complete. We will show this by reducing optimal binpacking [Garey, M.R. and Johnson, D.S. 79] to computing the parallel efficiency.

Lemma 13: Given an arbitrary complete interval tree *T*, it is NP complete to determine the parallel efficiency of *T*.

Proof: Let $\{w_1,...,w_n\}$ be an arbitrary sequence of packets. The bin packing problem consists of finding the minimal number of bins with capacity *c* that hold $\{w_1,...,w_n\}$. Given that sequence of packets, we construct a complete interval tree as follows: the leaves of tree are basic blocks $\{b_1,...,b_n,b_{n+1}\}$ such that the execution time for $b_i = w_i \forall i \in \{1,...,n\}$ and the execution time for $b_{n+1} = c$, such that $c > w_i, \forall i \in \{1,...,n\}$. The root of the complete interval tree consists of a node with an arbitrary execution time, w_{arb}. The minimal execution time for that tree is then $c + w_{arb}$, and the parallel efficiency is equal to 1 + (the minimal number of bins of capacity *c* needed to pack $\{w_1,...,w_n\}$). Hence, finding the parallel efficiency of arbitrary complete interval trees is NP complete.

If the execution time for every node in a complete interval tree is known, the length of the longest path from the root to a leaf node in the tree is also known. To process the complete interval tree in minimal time, there is a deadline for every node at which that node must be scheduled. Given the execution time and the deadline for every node, each node can be mapped to a 2 tuple $\{t_1,t_2\}$ such that t_1 is the node's deadline and $t_2 = t_1 +$ the node's execution time.

There are numerous ways to schedule the nodes in a complete interval tree such that the tree is processed in minimal time. A schedule consists of a mapping of nodes of the complete interval tree to a tuple that consists of processor and time, more formally:

Definition 14: *(Schedule)* Given a set of processors *P*, a complete interval tree with nodes *N*, and minimal execution time t_{min}, the function *S* with

$$S: N \times [0,t_{min}] \rightarrow (P \times [0,t_{min}])$$
$$S(n,t) = (p,t_n)$$

such that *n* is executable at time *t* and $t_n \leq d(n)$, where *d(n)* is the deadline for node *n* is called a schedule for *T*.

The simplest schedule that allows to process a complete interval tree in minimal time is obtained by scheduling every node as soon as it is executable. Thus, *all* basic blocks are scheduled at the beginning, each on a distinct processor. This leads to our first upper bound for the parallel efficiency of a complete interval tree, introduced in the next lemma.

Lemma 15: Given a complete interval tree *T* with *b* leaf nodes, the parallel efficiency of *T* is at most *b*.

Proof: By Lemma 12, $PARSET(t) \leq b \forall t$. Therefore, given *b* processors, there is always a distinct processor for each executable node in T, and T can be processed in minimal time. Therefore, b is an upper bound on the parallel efficiency of T.

It is easy to see that only for a complete interval tree T that is wide and flat, the number of leaf nodes is a reasonable approximation of T's parallel efficiency. For some trees, fewer processors are needed for minimal processing time when every node is scheduled at its *deadline*, rather than when it is executable. In such a schedule, the number of processors needed is equal to the number of nodes in the tree whose *execution interval* overlaps.

Definition 16: *(Execution interval of a node)* Given a node n in a complete interval tree with deadline t_d and execution time t_e, we say that we say that $[t_d, t_d+t_e]$ is the *execution interval* of n.

Lemma 17: Given a complete interval tree T with nodes N, the parallel efficiency of T is bounded by the largest subset $S \subseteq N$ such that the execution intervals of the nodes in S overlap.

Proof: Suppose that the size of the largest subset $S \subseteq N$ is p, but the parallel efficiency of T is $p' > p$. If every node in T is scheduled at its deadline, certainly T is processed in minimal time. Hence, there is a schedule using p processors in which T can be processed in minimal time. Hence, the parallel efficiency of T is at most p - a contradiction.

We have seen that it is hard to predict how many parallel processors can be used efficiently, even if implementation specific parameters are ignored. In a parallel implementation, system parameters can not be ignored, and even if a complete interval tree has a large parallel efficiency, in reality only a number of processors that is less than the parallel efficiency can be used effectively. We have implemented parallel interval analysis to assess how many processors can be used in parallel in practice.

3.5. Implementation: a test case

The goals of our implementation of parallel interval analysis were twofold. First, we wanted to assess the suitability of a parallelization based on a program's explicit loop structure. From the previous section it is clear that optimal parallel speedup for parallel interval analysis is hard to predict, even when system parameters are ignored. Finding a perfect schedule to process a given complete interval tree in parallel requires expensive analysis. Finding such a good schedule does not pay off in practice if the theoretically optimal speedup is cancelled by implementation and system parameters. Further, it is hard to predict the processing time of individual nodes in a complete interval tree accurately. Therefore, the second goal of our implementation was to assess whether a simple approach to scheduling the nodes of a complete interval tree suffices in practice.

3.5.1. Implementation details

Our program for parallel interval analysis runs as C process under the Mach operating system on an Encore Multimax system with 14 parallel processors [Multimax 88].

Even though our algorithm could be easily integrated into a production compiler, we wanted to study the effects of parallel interval analysis in isolation. We chose to build an interface to a production compiler to be able to obtain the flow graphs of real user programs without having to deal with the compiler's front end and back end.

The GNU C compiler developed at the Free Software Foundation [Stallman 88] served as "host compiler". A switch in the GNU C compiler causes the RTL representation of an input program to be written to a file. Reading the RTL representation from that file and re-constructing the program's flow graph makes our implementation completely modular: the basic blocks in the flow graph can be globally optimized, and the optimized RTL representation can be the input to the back end of the GNU C compiler in a separate phase. We therefore can focus on the parallelization of interval analysis but are able to combine our parallel implementation of interval analysis with (possibly parallelized) front ends and back ends of compilers whose intermediate representation is RTL. The possibility to run parallel interval analysis on any C program allows to assess the parallel performance for a realistic set of test cases. Parallelism is expressed by means of the cthreads library [Cooper 88].

3.5.2. The scheduling algorithm

The central scheduling construct of the parallel optimizer is a *ready queue*: each node of the complete interval tree that is ready to be processed is enqueued in the ready queue in arbitrary order. A two level hierarchy of threads is active during Stage3:

Master thread — The master thread has two functions: first, it sets up the ready queue (initially all leaves of the complete interval tree are scheduled) and forks a *fixed number* of *server threads*. Second, it is responsible for load balancing: the master thread enqueues the next node of the ready queue in the *data queue* of the server thread with the currently lowest work load.

Server threads — The server threads, each with its private *data queue*, execute concurrently. A server processes the nodes enqueued in its data queue and enqueues new nodes into the ready queue until it receives a termination signal from the master thread. Each node contains status information. Depending on that status information, the server applies either algorithm *I1* (Pass 1) or algorithm *I2* (Pass 2) to the node. During Pass 1, the server notifies the parent of the currently processed node about the completion of the child. If all children have finished the server enqueues the parent in the ready queue. During Pass 2, the server enqueues *all* children of the current node into the ready queue.

Each thread (the master thread and all server threads) runs on a separate processor. It should be noted that the master thread is merely a setup and load balancing agent and does not process any nodes of the complete interval tree. Therefore, the *best possible speedup* is *n-1* for *n* threads.

3.6. Results

Recall that our goal was to investigate how useful parallelism based on explicit program structure is in practice, so we measured the parallel speedup for the solution of only one data flow equation. Multiple pass optimizing compilers have to solve numerous data flow equations while the shape of the program remains the same. Fo a parallel implementation this means that a substantial part of the parallel overhead is independent of the number of equations solved. For multiple equations, task management, synchronization and scheduling efforts remain the same while the (sequential) work performed at the nodes of the complete interval tree is increased. As a result, better speedup can be anticipated if more equations are solved.

We therefore did not attempt to implement a tool that performs exhaustive data flow analysis in parallel but restricted our implementation to the availability system described in Section 2. By the previous argument, if we can observe good speedup for that one equation, our approach to parallelizing interval analysis can be successfully extended to global optimization problems that fit into the same framework.

3.6.1. The benchmark functions

The amount of parallelism in our implementation is determined by the shape of the complete interval tree which depends mainly on the size of the flow graph and the loop structure of the input program. We therefore used benchmark functions that differ in the number of basic blocks and contain nested loops, a characteristic that is met by many scientific programs. We used a set of five C benchmark functions from the scientific computing domain where the size of the flow graph varied between 30, 64, 176, 207 and 296 nodes (= basic blocks). In the following we will call those benchmark functions f_{30}, f_{64}, f_{176}, f_{207} and f_{296}.

3.6.2. Experiment description

We implemented both our parallelization and a standard implementation of sequential interval analysis based on the algorithm described in [Kennedy 81], orthogonal to the parallel implementation. It therefore provides a basis for a fair comparison with our implementation of parallel interval analysis.

For all functions, we measured both the parallel and the sequential execution time. For the parallel implementation, we varied the number of server threads (and therefore processors) between 2, 4, 6, 8 and 10. Recall that the master thread runs on a separate processor but is not involved in the work on the complete interval tree. So the number of processors varied between 3, 5, 7, 9 and 11 but the best possible speedup varies between 2, 4, 6, 8 and 10. To ensure that all parallel processors are dedicated to parallel interval analysis, we used the allocate_processor facility provided by the experimental Mach system running on our Encore Multimax. This facility allows to allocate a fixed number of processors for a given amount of time to a single user, possibly delaying the user until the system load allows the allocation. No other user can use those processors during that time interval. This facility minimizes the interference with other processes but does not eliminate it completely.

Each test was run multiple times. The numbers presented in this paper are the arithmetic mean of those measurements. The deviations of the individual measurements are within 5% of the average.

3.6.3. Measurements

We measured the speedup of parallel interval analysis over a standard sequential implementation of interval analysis. Before the server threads can start to execute in parallel, some setup work has to be carried out:

1. Initialization of the ready queue with all basic blocks
2. Allocation of processors

3. Forking of the server threads

In addition to the setup time and the time actually spent processing the nodes of the complete interval tree, the parallel execution time of parallel interval analysis accounts for task scheduling, task synchronization and load balancing. In the following, the total parallel execution time for parallel interval analysis is called $Total_{par}$. We measured the parallel setup time to perform steps 1.-3. and refer to it as O_{par}. Note that in our implementation, these steps are carried out sequentially; to speedup the setup time, steps 1., 2. and 3. could be carried out in parallel.

Our sequential equation solver uses the standard interval analysis algorithm and requires some implementation specific setup work before the tree nodes can be processed. In the following, this setup work of the sequential implementation is called O_{seq} and the total sequential time to perform Phase3 is called $Total_{seq}$.

Since we implemented interval analysis for only one data flow equation, the amount of work performed at each node of the complete interval tree is very small compared to a realistic optimizer. Therefore both the sequential setup time O_{seq} and the parallel setup time O_{par} account for an unrealistically large portion of $Total_{seq}$ and $Total_{par}$ respectively. To get a more accurate picture on the performance of our parallelization, we looked for a fair method to factor out both O_{par} and O_{seq} from the computation of the parallel speedup.

On first sight it sounds plausible to subtract the parallel setup time O_{par} from the total parallel execution time $Total_{par}$ and O_{seq} from $Total_{seq}$ respectively and use the resulting times to compute the parallel speedup. However, the parallel setup time is in general larger than the sequential setup time since it includes inherently expensive parts like for example processor allocation in a time shared multi user system. We therefore decided to account only for the sequential setup time by subtracting O_{seq} from both $Total_{par}$ and $Total_{seq}$ and obtain the speedup by dividing the two resulting numbers as shown in the following equation:

$$speedup = = \frac{Total_{seq} - O_{seq}}{Total_{par} - O_{seq}}$$

In the next section, the speedup observed always refers to the speedup introduced in the previous equation.

3.6.4. Speedup over sequential interval analysis

We were able to observe parallel speedup for every benchmark function. The speedup varied with the size of the benchmark function. Figure 3-11 depicts the speedup for f_{30} and f_{64}. Recall that the best possible speedup is at most *p-1* if we use *p* processors in parallel, since one of the processors is dedicated to the master thread. For both functions the speedup increases with increasing number of processors and decreases again when the number of processors exceeds 7 - an indication that scheduling overhead cancels the parallel speedup for small functions. In general the observed speedup is disappointing even though it increases for f_{64}. The situation changes drastically for f_{176} shown in Figure 3-12. The speedup increases almost linearly when the number of processors is small. We observe 5-fold speedup using 9 processors. Adding more processors results in a decrease in performance.

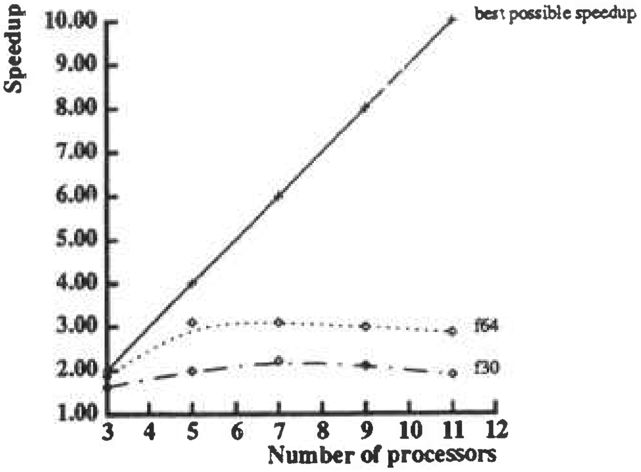


Figure 3-11: Speedup for small functions

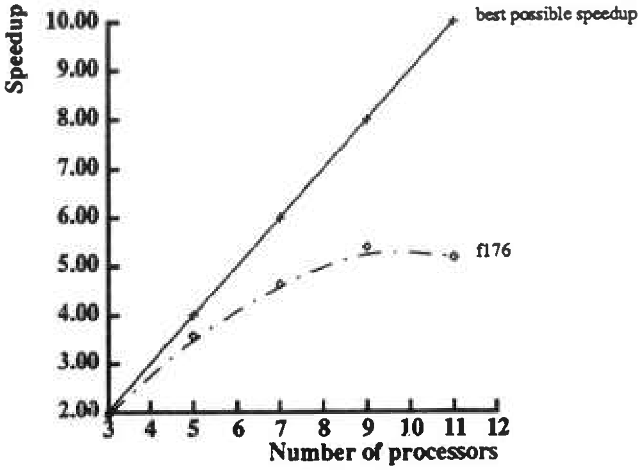


Figure 3-12: Speedup for medium function

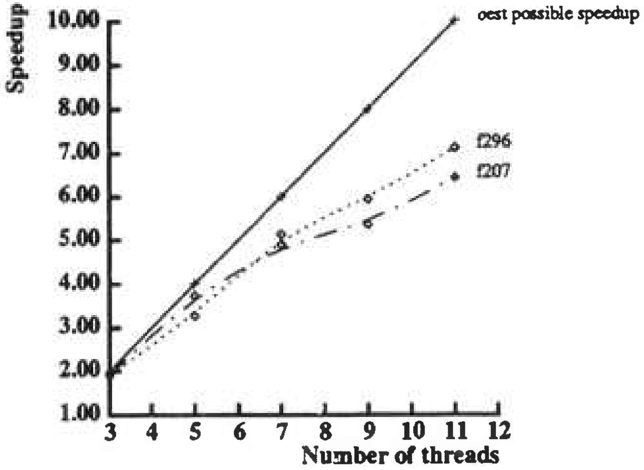


Figure 3-13: Speedup for large functions

Figure 3-13 depicts the speedup for f_{207} and f_{296}. Adding basic blocks helps: functions f_{207} and f_{296} show almost linear speedup and for both functions we do not have a "local speedup maximum", that is, the speedup increases steadily with increasing number of processors.

3.7. Discussion of the observed speedup

Our measurements show that for functions that are large enough, the speedup over sequential interval analysis is considerable. These results are particularly encouraging considering the simplicity of our implementation: the setup work is performed sequentially, the scheduling strategy is very simple, and we were still able to observe up to 7-fold speedup using no more than 11 processors, one of which is dedicated to load balancing and does not process any nodes of the complete interval tree. For all benchmark functions, the optimal theoretical speedup is much higher than the observed speedup. For instance, a lower bound of the parallel efficiency of the complete interval tree of our smallest example, f_{30}, was 10. In other words, when communication cost and system overhead are ignored, the parallel speedup should increase for each added processor up to 10 processors, and it should decrease when more than 10 processors execute in parallel. The observed speedup depicted in Figure 3-11 shows that the observed speedup decreases when more than 6 processors operate in parallel, so implementation and system overhead as well as communication cost cancel parallel speedup w' en too many processors operate in parallel. We observed the same phenomenon for our other benchmark functions and conclude that sophisticated methods to find a good or even optimal schedule for the nodes in a complete interval tree does not increase the observed speedup, and a simple scheduling strategy suffices for good practical results.

3.7.1. Possibilities to increase the speedup

There are two possibilities to increase the effectiveness of parallel interval analysis. First, better parallel speedup can be expected when the size of the input functions is increased. Second, implementation and system overhead decreases relative to the total execution time when the workload in the nodes of the complete interval tree is increased.

One way to increase the size of program flow graphs is to use procedure inlining if a program consists of many small functions. Thus, even when the size of the flow graphs is small, our approach can yield good speedup when combined with inlining, an important optimization in its own right. Further, many functions consist of more basic blocks than our benchmark functions, and for such large functions, better speedup can be expected.

Since we only implemented the solution of one data flow equation, the amount of work performed at each node of the complete interval tree smaller than in a typical global optimizer. Since the parallel overhead of parallel interval analysis is independent of the number of data flow equations even better speedup can be expected if the number of equations solved is increased.

3.7.2. Application spectrum for parallel interval analysis

The intervals of a flow graph correspond to the loop structure of the program. Working on (independent) intervals in parallel means that loops of the same loop nesting level are treated in parallel. Other global optimizations that allow to treat program loops on the same nesting level independently can be mapped into the same parallel framework. Examples are the extension of interval analysis to handle arrays reported in [GrossSteenkiste 90] and other methods for data dependence analysis used in vectorizing compilers. The fact that we parallelized the solution of an availability system does not restrict our approach to just data flow equations. Our measurements indicate that parallelism based on program structure can yield significant speedup - independent of the specific optimization problem that is solved in parallel.

3.8. Related work

Other parallel algorithms for global data flow algorithms are reported in [LeeMarloweRyder 91] and [GuptaPollockSoffa 90]. The parallel hybrid algorithm described in [LeeMarloweRyder 91] combines the iterative technique with a structured method for data flow analysis. Maximal strongly connected components in the flow graph are processed in parallel. The iterative method is used to compute local data flow information for each component, and in a subsequent propagation phase global data flow information is computed for each component. The communication between parallel processes in this approach is greater than the communication required for parallel interval analysis. In the measurements of the parallel speedup, the number of processors that execute concurrently is varied between 2 and 8. The best parallel speedup reported in that paper is less than 5.

Goal of the approach taken in [GuptaPollockSoffa 90] is to partition a program flow graph independently of program structure. A program is decomposed into single-entry-single-exit regions that can be smaller than intervals, therefore the size of the parallel tasks can be chosen more evenly. No implementation or parallel speedups are reported. This method has the potential to create more parallelism in global data flow analysis, but it is not clear whether this parallelism can be taken advantage of in an implementation. Our own measurements indicate that even the coarser grained parallelism in parallel interval analysis is cancelled by system and implementation parameters when too many processors execute concurrently.

3.9. Chapter summary

Our parallelization of interval analysis is based on explicit program structure. Data locality is given by the loop structure of the input program and can be exploited for the parallelization in a straightforward manner. Basis of our parallel implementation is the complete interval tree which captures the interval partitions of a program. It is NP complete to compute a lower bound on the number of processors needed to process a given input program in minimal time, but we gave some simple approximations of this lower bound. Our implementation used a very simple scheduling algorithm for the nodes in the complete interval tree, and we were able to observe considerable parallel speedup. Due to system and implementation overhead, the speedup declined when too many processors operated concurrently. In all cases the number of processors for which the parallel speedup declined was smaller than a lower bound on the number of processors needed to process a complete interval tree in minimal time. We conclude that a straightforward parallelization based on the loop structure of a program works well in practice, and that complicated scheduling algorithms or efforts to create finer grained parallelism do not result in increased observed speedup.

Chapter 4

Global register allocation: background

4.1. Introduction

In the previous chapter we showed a simple parallelization of global data flow analysis that was based on explicit program structure. Data locality was given by the loop structure of the input program and the results for an inner loop could be embedded into the results for an enclosing loop without backtracking. A number of global compiler optimizations do not fit into this framework because in general it is not possible to embed a partial solution of one program construct into the enclosing loop or conditional without backtracking. We call such compiler optimizations *unstructured* optimizations. An example of such an optimization is global register allocation. Global register allocation is the problem of mapping variables that live across basic block boundaries to machine registers. The problem of optimal global register allocation is mathematically equivalent to the problem of finding an optimal coloring for an undirected graph representing register conflicts. Global register allocation via graph coloring is used in a variety of compilers. In general it is assumed that the graph denoting the register conflicts of a program is an arbitrary graph. Finding an optimal coloring of arbitrary graphs is NP complete.

Given a fixed number of processors, it is very difficult to come up with a straightforward parallelization of an NP complete problem. The research goal for the second part of this thesis is to investigate whether knowledge about program structure can be used to *implicitly* guide the parallelization of global register allocation. In other words, we examine whether the knowledge about program structure can be used to analyze the graph that represents register conflicts such that points can be found at which the graph can be partitioned into independent components. The partitioning of the conflict graph into independent components is based on *clique separators*, first introduced in [Tarjan 85]. The problem is that for arbitrary graphs it is very difficult to detect clique separators.

The first step of our research is to establish a connection between program structure and register conflict graphs. We propose a model in which structural knowledge about a program is encoded in the register conflict graph. We then use this knowledge to deduce characteristics of the partial register conflict graph of individual loops and conditionals that are later used to detect clique separators in the conflict graph.

This chapter provides the background for our model. We first give some basic definitions and the description of our input model. We then introduce the standard method for register allocation via graph coloring, followed by an example of a conflict graph that can not be colored with the standard method. We conclude the chapter by classifying the live ranges that occur in loops and conditionals.

4.2. Global register allocation: basic definitions

Registers are a scarce resource on all modern computer architectures [Pat/Hen 90]. Optimizations like common subexpression elimination, constant propagation etc. create temporaries that can be used most efficiently if they reside in registers. Once a compiler has decided which variables and temporaries are to be placed into registers, register allocation is the problem of mapping machine registers to variables and temporaries such that runtime efficiency of the machine code is maximized. In general, variables and temporaries are not used in every basic block of the program flow graph, but have *local* or *global live ranges*.

> **Definition 1:** *(Local live range)* A local live range of a variable v is a sequence of instructions $i_1, \dots ,i_n$ such that i_1 is a definition of v, and i_n is the last use of v before a re-definition of v. Further, all instructions of that sequence occur in the same basic block.

In general global variables or temporaries created by global optimizations are live across basic block boundaries. Global data flow analysis determines for each basic block b which variables and temporaries are live at the entry to the basic block [Aho 84]. Similar to local live ranges that consist of sequences of instructions, global live ranges consist of sets of basic blocks. Because the exact order of instructions inside basic blocks is usually not known at the time of global register allocation, we assume that global live ranges extend throughout the instructions of every basic block that is part of the live range. So in our model, global live ranges start and stop at basic block boundaries. Our definition of a global live range is based on the *live range graph of a variable*, defined below.

> **Definition 2:** *(Live range graph of a variable)* Given the flow graph of a program consisting of basic blocks B, the live range graph of variable a consists of vertices V and directed edges E. The set of vertices of the live range graph is defined as $v \in V$ iff $v \in B$ and a is live in v. The edges of the live range graph are defined as $e=(v_1,v_2) \in E$ iff there is an edge from v_1 to v_2 in the flow graph and any definition of a in v_2 is preceded by a use of a in v_2.

The live range graph of a variable is the basis for our computation of live ranges:

> **Definition 3:** *(Live ranges of a variable)* The live ranges of a variable a are the connected components of the live range graph of a.

In other words, the live range of a variable is a contiguous set of basic blocks of the flow graph in which the variable lives. A variable can have several independent live ranges. If variable a lives in basic blocks b_1 and b_2, and b_1 and b_2 are not part of the same live range, the variable can reside in different registers in b_1 and b_2 respectively.

Figure 4-1 shows both a flow graph and the live range graph for variable a.

Variable a is live in basic blocks *B1*, *B2*, *B4*, *B5*, the split node and *B6*. The live range graph for a is depicted to the right. Because there is a definition of a in *B4* that is not preceded by a use of a, there is no edge from *B2* to *B4* in the live range graph. For the same reason, there is no edge between *B4* and *B5* in the live range graph. The connected components of the live range graph are *{B1,B2}*, *{B4}*, *{B5,B6}*. Therefore, a has three distinct live ranges. The live range that consists of *B4* is a local live range, the other two are global live ranges.

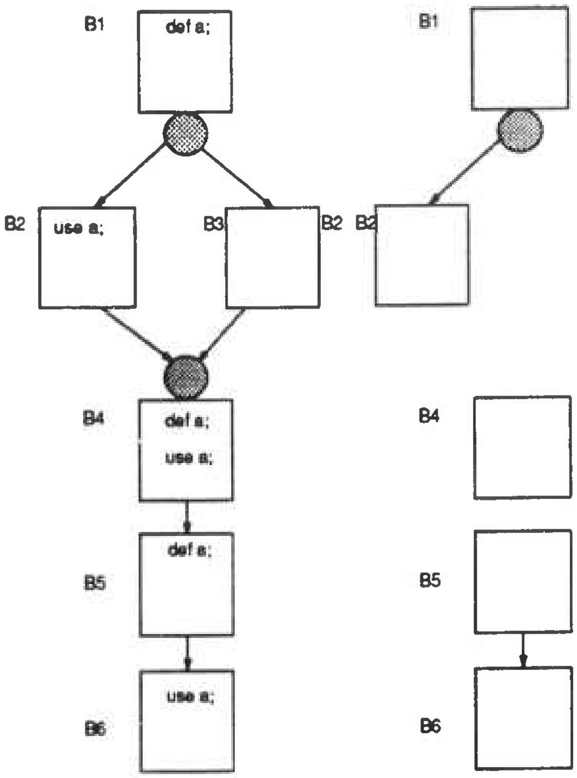


Figure 4-1: Sample live range

Local live ranges and global live ranges are mapped to machine registers in different phases - local register allocation and global register allocation respectively. Sometimes, local register allocation is performed fairly late in the compilation process, because some instruction reordering inside basic blocks might be done by the code scheduler. Further, some compilers assume dedicated registers for global live ranges.

One major difference between local register allocation and global register allocation is that local register allocation is performed for live ranges that consist of instructions that form straight line code. We will see that optimal global register allocation can be solved efficiently for straight line code. Local register allocation is often done at code selection time, and is in that case a non trivial task.

Because global live ranges consist of sets of basic blocks that may be part of loops or conditionals, the register conflict graphs for global live ranges are more complex, and finding an optimal coloring for those graphs is NP complete.

The research focus of this thesis is on global register allocation. Before we introduce our model of structural global register allocation, we introduce register conflict graphs and the standard method for global register allocation via graph coloring.

4.2.1. Standard method for global register allocation by graph coloring

The standard method for global register allocation via graph coloring operates on the *register conflict graph*. Given a flow graph and for each basic block in the flow graph the set of variables and tempo:aries that are live in the basic block, the standard register conflict graph is defined as follows:

> **Definition 4:** *(Standard register conflict graph)* A standard register conflict graph $G=(V,E)$ consists of vertices N and edges E. Each $n \in N$ corresponds to a global live range. Two vertices n_1, n_2 are connected by an edge e iff there is at least one basic block b such that b is part of both live ranges n_1 and n_2.
>
> In other words, n_1 and n_2 may co-exist during program execution and hence must be placed in different registers.
>
> **Definition 5:** *(Chromatic number of a graph)* The chromatic number of a graph G is the minimal number of colors needed to color all nodes in G such that no two adjacent nodes have the same color.

The most commonly used method to color register conflict graphs was introduced in [Chaitin 81] and works as follows. Let k be the number of available machine registers and G be the register conflict graph. A k-coloring of G based on the following observation: if G has a node n with less than k adjacent nodes, G is k-colorable iff the graph G' obtained by removing n and all edges incident on n is k-colorable. Thus, all nodes of degree less than k are removed from G until G' is empty or consists only of nodes with degree greater than k.

If G' is not empty and no nodes can be removed from G', the standard method assumes that no k-coloring exists for G, and that therefore not all live ranges can reside in registers at all times. The basic idea to solve this problem is to remove nodes from G' and assume that the corresponding live ranges are stored in memc ather than register. The node removal process is repeated on the altered graph until a k-coloring is found.

This algorithm is used for many implementations of global register allocation [HilLar 86, Wall 86, ChowHen 90]. In the following we will refer to this coloring method as the *node removal technique*.

4.2.2. Shortcomings of the node removal technique

In the standard node removal technique, the decision of which node is removed next is guided by the number of outgoing edges. Two nodes n_1 and n_2 that are removed from the graph in sequence might correspond to live ranges of unrelated parts of the flow graph - the program structure does not play a role in the overall coloring process. Further, there exist register conflict graphs that are k-colorable, but in which each node has at least k neighbors, so the node removal technique is not able to produce a k-coloring. See for instance the graph depicted in Figure 4-2.

Figure 4-2 shows a register conflict graph in which each node has at least 3 neighbors. Given the number of available register is 3, the node removal technique decides that at least one of the nodes must be spilled to memory. As shown in the Figure, a 3-coloring for the graph exists.

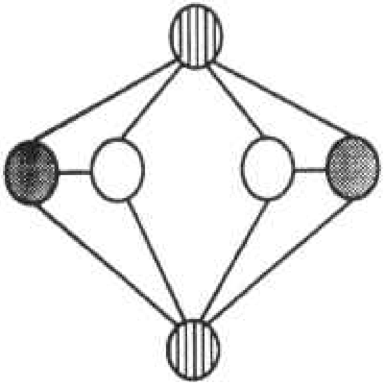


Figure 4-2: Example where standard method fails to come up with a *k*-coloring

In our approach to global register allocation, we use knowledge about the program structure to guide the register allocation process. We will see later that there are some cases in which the structural knowledge encoded in the register conflict graph can be used to produce *k*-colorings where the node removal technique is unable to do so.

We now discuss our classification of live range as they occur in straight line code, loops and conditionals. Based on these classifications, we show some properties of register conflict graphs for each programming construct in the following chapters.

4.2.3. Continuous and broken live ranges

We partition the live ranges of variables into two groups: *continuous* and *broken* live ranges.

Definition 6: *(Broken live range)* Given a variable v with live range l that consists of basic blocks $\{b_1,\ldots,b_n\}$, we say that l is broken iff there are two basic blocks b_i and b_j both in $\{b_1,\ldots,b_n\}$ that meet the following condition: There is a backarc free path from b_i to b_j that contains a basic block b_k such that b_k contains a definition of v and every use of v in b_k is preceded by that definition.

Definition 7: *(Continuous live range)* A live range is continuous iff it is not broken.

Examples of broken and a continuous live ranges are given in Figure 4-3. In the loop labeled A, variable v lives in basic blocks 1,3 and 4. Because there is a backarc free path between blocks 1 and 3 and 3 contains a definition of v, the live range for v is broken. The conditional labeled B also contains a broken live range for variable v - there is a backarc free path from block 1 to block 3 and there is a definition of v in block 3. Hence, the live range for v is broken. The live range of v depicted in the flow graph labeled C is continuous. Even though the live range of v in the flow graph labeled D contains *all* basic blocks of that flow graph it is a broken live range because the backarc free path between blocks 2 and 4 contains block 3 which contains a definition of v.

Intuitively, broken live ranges can contain "holes" - holes in a broken live range consist of basic blocks that are not in the live range but "between" basic blocks that form the live range. More formally, the definition of a hole is based on an undirected graph which we call *hole graph.*

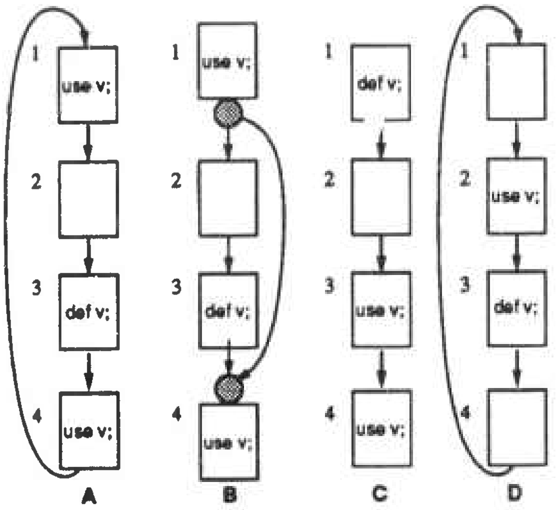


Figure 4-3: Examples of broken and continuous live ranges

> **Definition 8:** *(Hole graph of a live range)* Given a flow graph F with basic blocks B and a live range l, the hole graph of l is the graph induced by the set of nodes $\{B-l\}$ and all edges of F except F's backedges.

In other words, the hole graph of a live range l is derived from the flow graph by removing all basic blocks that form l, all edges incident on basic blocks in l and all backarcs. The connected components of a hole graph of a live range are called *holes* of that live range provided these connected components are "between" basic blocks that are part of the live range. In other words, there must be a backarc free path from the live range that leads into the hole, and there must be a backarc free path from the hole that leads into the live range. More formally:

> **Definition 9:** *(Hole of a live range)* Given a flow graph F and a live range l consisting of basic blocks $\{b_1,...,b_n\}$, leg H be the hole graph of l. A connected component c of H is called a hole of l iff $\exists b_c \in c$ such that
>
> 1. $\exists b_i \in \{b_1,...,b_n\}$ such that there is a backarc free path from b_i to b_c
>
> 2. $\exists b_j \in \{b_1,...,b_n\}$ such that there is a backarc free path from b_c to b_j

Figure 4-4 gives an example of a broken live range and its hole graph. In that example, the flow graph to the left consists of a loop. Variable a has a broken live range consisting of basic blocks $\{1,2,7,8\}$. The hole graph is derived from the loop by removing those basic blocks and all backedges as well as edges to and from $\{1,2,7,8\}$. The hole graph consists of one connected component. Hence, the live range for a has only one hole, which consists of all basic blocks in the hole graph.

Some broken live ranges do not contain any holes - even though there is a re-definition in some basic block that is part of the broken live range, the hole graph of that live range can be empty. In that case, the broken live range can be treated as if it were continuous. We call such live ranges *continuous equivalent.* The live range depicted in the flow graph labeled D in Figure 4-3 is an example of a continuous equivalent live range.

> **Definition 10:** *(Continuous equivalent live range)* We say that a broken live range l is continuous equivalent iff l contains no holes.

One nice property of continuous equivalent live ranges is that for each continuous equivalent live range there exists an equivalent continuous live range. What sounds like a play of words will be formally shown in the next lemma.

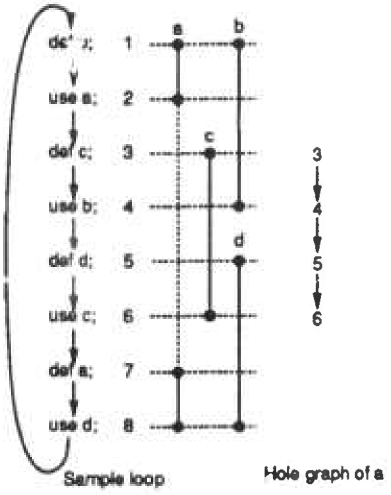


Figure 4-4: Hole graph and hole of a broken live range

Lemma 11: Given a register conflict graph G that contains a continuous equivalent live range l for a variable v, let $\{b_1,...,b_m\}$ be the set of basic blocks that contain a definition of v that is not preceded by a use of v in the same basic block. The live range does not change if a use of v is inserted into every $b_i \in \{b_1,...,b_m\}$.

Proof: Because l contains no holes, no backarc free path between any two basic blocks in l can contain a basic block that is not part of l. In particular, every path to a basic block that contains a definition of v consists of basic blocks that lie entirely in l. Hence, adding a use of v at the top of the basic blocks that define v is not going to change l.

By adding uses of v to the defining blocks that are part of a continuous equivalent live range for variable v, the broken live range is changed into a continuous live range. Hence, for the purpose of register allocation, continuous equivalent live ranges can be turned into continuous live ranges by modifying the program slightly. Like broken live ranges, continuous equivalent live ranges are caused by re-definitions in loops or conditionals and it must be ensured that both parts of a continuous equivalent live range end up in the same register. Hence changing a continuous equivalent live range into an equivalent continuous live range has no influence on the subsequent graph coloring.

A program that consists entirely of straight line code can only contain continuous live ranges, stated formally in the next lemma.

Lemma 12: Given a piece of straight line code consisting of basic blocks $\{b_1,...,b_n\}$, let l be a live range that consists of basic blocks that form a subset of $\{b_1,...,b_n\}$. Then, l is continuous.

Proof: Given a piece of straight line code $\{b_1,...,b_n\}$, there is at most one path between any two blocks $b_i,b_j \in \{b_1,...,b_n\}$. Given a live range l for a variable v, we show by contradiction that l must be continuous.

Let l be a broken live range for variable v consisting of basic blocks that form a subset of $\{b_1,...,b_n\}$. By definition of broken live ranges, there must be two basic blocks b_i and b_j that are part of l, and there must be a path from b_i to b_j that contains a basic block b_k that contains a definition of v which is not preceded by a use of v in b_k. By definition of live range graphs, there can be no edge between b_k and a basic block preceding b_k. Because the flow graph contains no backarcs, b_k can not be part of the same connected component as b_i, since b_i must precede b_k. Because b_k precedes b_j, b_j can not be part of the same connected component as b_i - a contradiction. Therefore, l must be continuous.

By Lemma 12 we know that in well structured programs only loops and conditionals can contain re-definitions that cause a hole in a live range. Holes in a live range for a variable v occur if several definitions of v are used in the same basic block. A loop can contain a re-definition of a variable v at the bottom of the loop that is used at the top of the loop. A hole that is caused by such a re-definition consists of the basic blocks "between" the basic block that contains the use and the basic block that contains the re-definition. An example of this situation is given in Figure 4-5. The figure shows both a nested loop and the interval representation of the register conflict graph. The live range for variable a contains a hole consisting of basic blocks {3,4,5,6}. The hole of a is caused by a re-definition of a in basic block 7, which is part of the innermost loop. Basic block 2, which uses that re-definition of a, is part of the outer loop, but not of the inner loop. Basic blocks between the use and the re-definition are {3,4,5,6} - the basic blocks that form the hole.

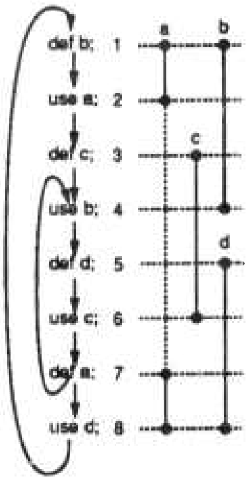


Figure 4-5: Hole caused by a re-definition inside a loop

A conditional can contain one branch clause c with a re-definition of a variable v with that is live in both the split node and the join node of the conditional. Hence, there must be a definition of v in a basic block that occurs on the path from the program entry to the split node and a use of v in a basic block b that is dominated by the join node. Hence, both definitions of v in b and in the branch clause c are used outside the conditional. An example of a conditional that contains a re-definition of such a live range is shown in Figure 4-6.

The live range for variable g consists of basic blocks {1,2,5,6,7}. The hole graph for g consists of basic blocks 3 and 4 and is depicted to the right. Because there is a backarc free path from basic block 2 to basic block 3 and there is a backarc free path from basic block 4 to basic block 7, blocks {3,4} constitute a hole of the live range for g. That hole is caused by the re-definition of g in basic block 5 which is part of one branch clause. Both definitions of g in basic blocks 1 and 5 are used in basic block 7.

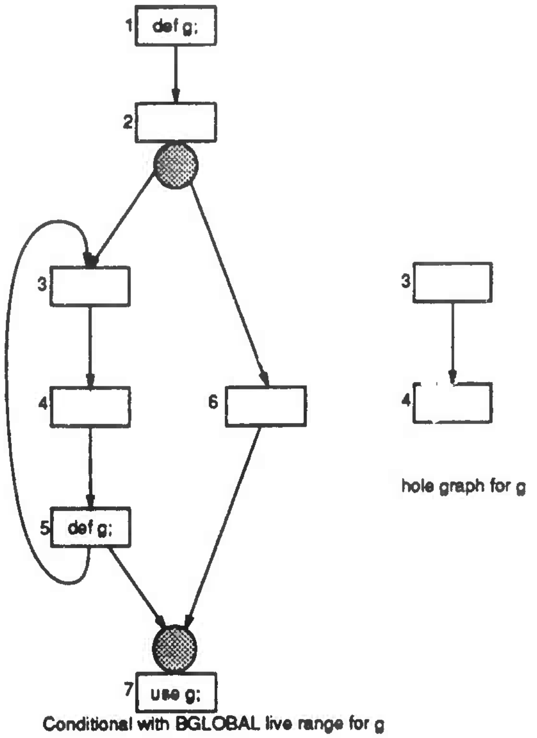


Figure 4-6: Hole caused by re-definition of a variable with *BGLOBAL* live range in a conditional

A hole of a live range can share basic blocks with more than one programming construct. In the example depicted in Figure 4-6, the hole of variable g's live range contains basic blocks {3,4} which are both part of the loop consisting of basic blocks {3,4,5} and of the entire conditional. Given the live range for g, {1,2,5,6,7}, the fact that the basic blocks "between" 2 and 5 are missing makes {3,4} a hole. Looking at the loop in isolation, the live range of g consists only of basic block 5 and therefore basic blocks 3 and 4 are not recognized as a hole of g's live range. Considering the conditional in isolation (that is, basic blocks {2,3,4,5,6,7}), the live range of g consists of basic blocks {2,5,6,7}. Note that the hole of g's live range exists in the conditional, even when the rest of the flow graph is ignored. Hence, the hole of the live range for g can be "linked" to the conditional but not the loop nested inside the conditional.

The reason why we want to link a hole to a *particular* loop or conditional is that the exact position of the hole in the program flow graph determines which simplifications can be carried out on the flow graph such that the register conflict graph is unchanged by those simplifications.

Given the set of loops and conditionals in a well structured program that share basic blocks with a given hole h for a broken live range l, we link h to the innermost programming construct that contains a subset of l such that h is a hole of this subset. In our previous example (Figure 4-6), the conditional is the innermost programming construct that contains a subset g' of the live range for g such that g' is broken and has the same hole as g. Now more formally:

Definition 13: *(Linking of a hole)* Given a well structured flow graph F that contains a broken live range $l=\{l_1,...,l_n\}$ that contains a hole $h=\{h_1,...,h_h\}$, let P be the set of loops and conditionals in F such that each loop or conditional in P contains at least one basic block in

$\{h_1,...,h_h\}$. For $p \in P$ let $l_p = l \cap p$, that is the intersection of the basic blocks that form p and l. We say that h is linked to p iff

1. h is a hole of l_p
2. for every loop or conditional p' nested inside of p, h is not a hole of p'.

In our example depicted in Figure 4-5, the hole of variable a is linked to the outer loop, and in the example depicted in Figure 4-6, the hole of the live range for g is linked to the outer conditional.

4.3. Live ranges in loops

The intuitive background of our classification of live ranges in loops is the impact they have on register conflict graphs for loops. We partition the live ranges in loops into *backarc* and *forward* live ranges

More formally:

Definition 14: *(Forward live range)* A global live range $b_1,...,b_n$ of a variable v is called a *forward live range* of a loop consisting of basic blocks $\{l_1,...,l_m\}$ iff

1. $\{b_1,...,b_n\} \cap \{l_1,...,l_m\} \neq \emptyset$
2. all paths from basic blocks $b_{def} \in \{b_1,...,b_n\} \cap \{l_1,...,l_m\}$ that contain a definition of v to $b_{use} \in \{b_1,...,b_n\} \cap \{l_1,...,l_m\}$ that use the definition in b_{def} are backarc free.

The complement of forward live ranges are *backarc live ranges*, defined as follows:

Definition 15: *(Backarc live range)* A global live range $b_1,...,b_n$ of a variable v is called a *backarc live range* of a loop consisting of basic blocks $\{l_1,...,l_m\}$ iff

1. $\{b_1,...,b_n\} \cap \{l_1,...,l_m\} \neq \emptyset$
2. at least one path from a basic block $b_{def} \in \{b_1,...,b_n\} \cap \{l_1,...,l_m\}$ that contains a definition of v to $b_{use} \in \{b_1,...,b_n\} \cap \{l_1,...,l_m\}$ that uses the definition in b_{def} contains a backarc.

Figure 4-7 depicts loop constructs with a forward live range and a backarc live range respectively. The live range for a is a forward live range, because it is defined in *B1*, and there is no backarc on the path from *B1* to *B2*, where a is used. The live range for b consisting of *B1, B2, B3* is a backarc live range, because both uses of b in *B1* and *B2* can only be reached via a backarc from the basic block that contains the definition of b. Note that the live range for b is continuous - it contains all basic blocks that form the loop.

The backarc live range depicted in Figure 4-8 is not continuous. The live range for c consists of blocks *B4* and *B1*, and the path from the defining block *B4* to the use in *B1* contains a backarc. Note that the loop contains two more basic blocks, *B2* and *B3* that form a hole of the live range for c. Hence, the register for c can be used for a different live range that consists of *B2* and *B3*. The live range for c contains a hole that consists of basic blocks *B2* and *B3*. Further, the hole in the live range is caused by a definition of c in block *B4*.

A third example of a backarc live range in a loop is depicted in Figure 4-9. In that example, variable c's live range consists of basic blocks $\{B1,B2,B4,B5\}$ - a broken live range. The hole of the live range for c is linked to the conditional with split node *B2* and join node *B5*. Even though the live range for c is broken, we call it a *Loop-continuous backarc* live range, because the hole is not linked to the loop.

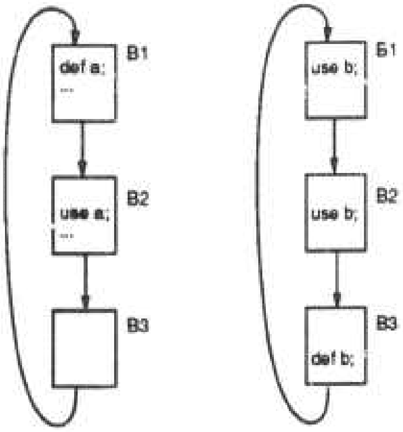


Figure 4-7: Examples of backarc and forward live ranges

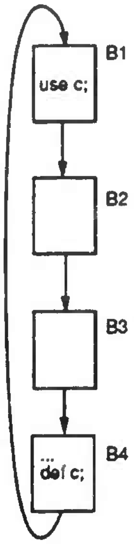


Figure 4-8: Another backarc live range

Definition 16: *(Loop-continuous live range)* A live range *l* of variable *v* in a loop *L* with head *h* and exit *e* is called a *loop-continuous live range* iff there is no hole *h* of *l* such that *h* is linked to *L*.

Definition 17: *(Loop-broken backarc live range)* A backarc live range that is not loop-continuous is called a loop-broken backarc live range.

Figure 4-10 shows examples of loop-broken and loop continuous backarc live ranges.

The live range for variable *c* depicted to the left consists of {*B1*,*B2*,*B4*}. Because the hole consisting of *B3* is linked to the outer loop, the live range for *c* is a loop-broken live range. Adding *B3* to the live range for *c* changes it into a loop-continuous live range, shown in the middle of Figure 4-10. The example depicted to the right shows another example of a live range that is loop broken: the hole consisting of *B2* and the join node is not entirely contained in the inner conditional. It is easy to see that a loop-broken live range must be a backarc live range. A live range that contains a hole but is not loop-broken must be a forward live range - the hole is linked to a loop or a conditional nested inside the loop.

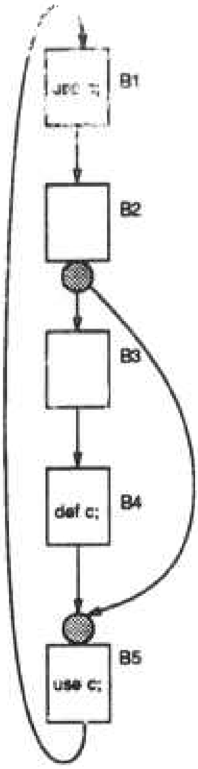


Figure 4-9: A continuous backarc live range

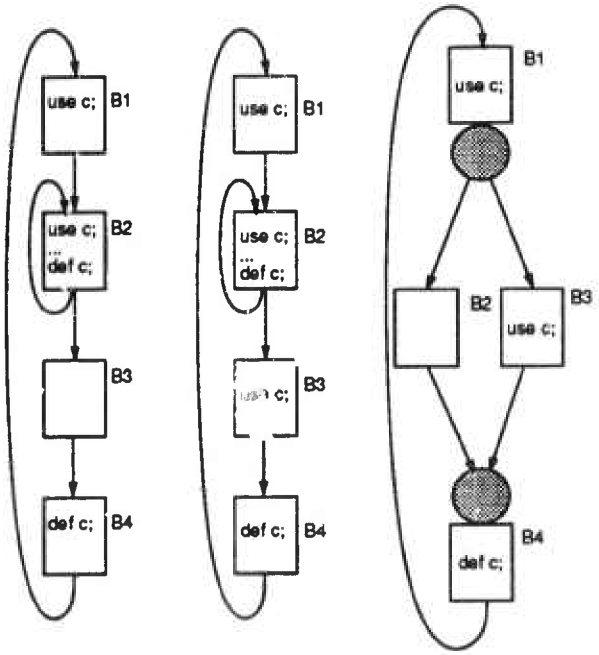


Figure 4-10: Examples of loop-broken and loop-continuous live ranges

The class of register conflict graphs for loops that contain broken live ranges contains arbitrary circular arc graphs, for which optimal coloring is NP complete, which will be shown in Chapter 5. We will see that only loops that contain broken live ranges can cause arbitrary register conflict graphs. This is not generally the case for conditionals. The types of live ranges in conditional branches are discussed next.

4.4. Live ranges in conditionals

We partition the live ranges that occur in conditionals into four classes, *LOEN*, *LOEX*, *BLOCAL* and *BGLOBAL*. The basis for this classification is the presence or absence of the split and join node in the live ranges. The intuitive reason for our classification is that it permits to detect dependencies between the register conflict graphs of individual branch clauses.

If each live range in a conditional is "local" to one dist[illegible] branch clause, the conflict graphs for the branch clauses are independent. Therefore, for each [illegible] clause the register conflict graph can be colored independently. On the other hand, live ranges that contain the split or join node of a conditional are shared by the register conflict graphs of individual branch clauses and they are no longer independent. We will see in Chapter 5 that dependencies between the conflict graphs of individual branch clauses can lead to overall conflict graphs that are hard to color optimally.

We formalize these characterizations and partition live ranges that occur in a branch construct into four classes:

> **Definition 18:** *(LOEN live range)* A live range is called a *LOEN* (Life On ENtry) live range in a conditional with split node *S* and join node *J* iff it contains at least one basic block that is part of a branch clause and *S* but not *J*.
>
> **Definition 19:** *(LOEX live range)* A live range is called a *LOEX* (Live On EXit) live range in a conditional with split node *S* and join node *J* iff it contains at least one basic block that is part of a branch clause and *J* but not *S*.
>
> **Definition 20:** *(BGLOBAL live range)* A live range is called a *BGLOBAL* live range in a conditional with split node *S* and join node *J* iff it contains at least one basic block that is part of a branch clause and both *S* and *J*.
>
> **Definition 21:** *(BLOCAL live range)* A live range is called a *BLOCAL* live range in a conditional with split node *S* and join node *J* iff it contains basic blocks of a branch clause but not *S* or *J*.

In the example depicted in Figure 4-11, *g* is *BGLOBAL*, *a* is *LOEN*, *d* is *LOEX* and both *b* and *c* are *BLOCAL*.

Analogous to our definition of loop-continuous and loop-broken live ranges, there are conditional-continuous and conditional-broken live ranges. A conditional-continuous live range can have a hole that is not linked to the conditional itself, but instead to a conditional or loop nested inside. This is depicted in Figure 4-12.

The figure shows two nested conditionals. The outer conditional with split node 1 and join node 6 contains a *BGLOBAL* live range for *v*, which consists of {1,2,4,5,6}. This live range is broken, and contains a hole formed by block 3. This hole is caused by a re-definition of *v* in basic block 4, which is part of the inner conditional with split node 2 and join node 5. Like for loops, the live range for *v* is therefore conditional continuous in the outer conditional, and conditional-broken in the inner conditional. A formal definition follows.

> **Definition 22:** *(Conditional-continuous live range)* A live range *l* for a variable *v* with a hole *h* is *conditional-continuous* in a conditional *C* with split node *S* and join node *J* iff there is no hole *h* of *l* such that *h* is linked to *C*.

The complement to a conditional-continuous live range is a conditional broken live range.

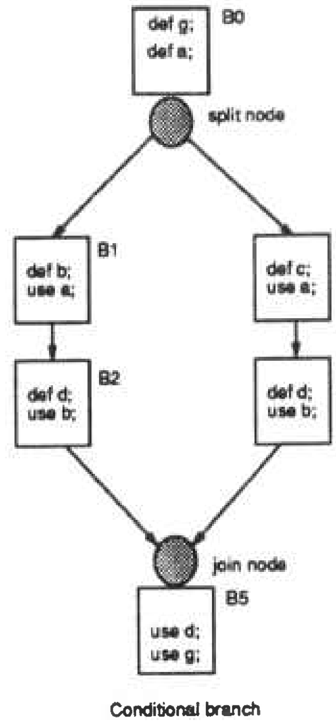


Figure 4-11: Types of live ranges in a conditional

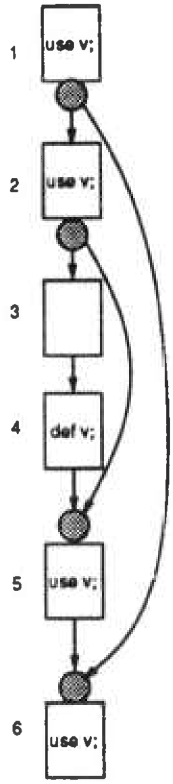


Figure 4-12: Conditional-continuous and conditional-broken live range

Definition 23: *(Conditional-broken live range)* A live range l for a variable v is *conditional-broken* in a conditional C iff it is not conditional-continuous in C.

Note that a conditional-broken live range must be a *BGLOBAL* live range. Broken *LOEN*, *LOEX* or *BLOCAL* live ranges in a conditional *C* must be conditional-continuous in *C* - holes in such live ranges must be entirely contained in programming constructs nested inside *C*.

4.5. Chapter summary

We have classified the live ranges of well structured programs. Live ranges can be continuous or broken. Live ranges in straight line code are all continuous, while conditionals and loops can contain re-definitions that cause a live range to be broken. Loops can contain forward live ranges and backarc live ranges. Live ranges in loops can be loop-continuous or loop-broken. Conditionals can contain *BLOCAL*, *BGLOBAL*, *LOEN* and *LOEX* live ranges. A live range *l* in a conditional can be conditional-continuous or conditional-broken, depending on which programming construct causes a hole in *l*.

In the next chapters we show how this classification can be used to analyze the shape of register conflict graphs for loops and conditionals.

Chapter 5

Register conflict graphs for compound programming constructs

In this chapter we characterize properties of register conflict graphs of straight line code, loops and conditional branches. The first part of this chapter is devoted to re-stating some properties of register conflict graphs that have been established before [Gol 85, Fishburn 85, Bernstein et al 89]. We first showing that register conflict graphs for straight line code are interval graphs and show that the standard node removal technique is able to produce an optimal coloring for interval graphs.

The class of register conflict graphs for loops and conditionals contain arbitrary circular arc graphs for which finding an optimal coloring is NP complete. When certain restrictions are met by the live ranges that occur in a conditional or a loop, the register conflict graph of that conditional or loop is equivalent to the register conflict graph of straight line code. One contribution of our structured model for global register allocation is that it enables us to establish such restrictions systematically. The bulk of this chapter is devoted to the description of situations in which it is possible to produce an optimal coloring for register conflict graphs of loops and conditionals in polynomial time. In particular we discuss cases in which it is possible to simplify conditionals and loops without altering the register conflict graph. Goal is to create straight line code that is equivalent to loops and conditionals for the purpose of register allocation. We will see that the simplifications to obtain straight line code enable us to locate clique separators in the register conflict graph.

5.1. Register conflict graphs for straight line code

We first turn our attention to register conflict graphs of straight line code. We show how the live ranges in a register conflict graph for straight line code can be mapped to intervals on the real line. We then show how the cliques in the register conflict graphs can be ordered to form a sequence of cliques, and use that to show that the node removal technique is able to produce an optimal coloring in polynomial time.

We will show that register conflict graphs of straight line code are interval graphs by mapping every live range in the flow graph of straight line code to an interval on the real line. The first step is to define a monotonously increasing function f: $\{b_1,...,b_n\} \rightarrow \mathbf{R}$ as follows: $f(b_i)=i$, $i \in \{1,..,n\}$. Note that f is both monotonely increasing and bijective.

Definition 1: *(Interval on the real line)* An interval on the real line $[n_1,n_2],n_1 < n_2$, is a subset of **R** such that $n_1 \in \mathbf{R}$ and $n_2 \in \mathbf{R}$ and $\forall \{x|n_1 \leq x \leq n_2\} x \in [n_1,n_2]$ and $\forall \{y|y<n_1\}\ y \notin [n_1,n_2]$ and $\forall \{y|y>n_2\}\ y \notin [n_1,n_2]$.

Definition 2: *(Interval graph)* An *interval graph* is a graph whose vertices v can be represented by intervals I_v of the real line such that two vertices are adjacent if and only if the corresponding intervals intersect.

Because the live ranges in straight line code are contiguous sets of basic blocks, we can map a straight line live range $\{b_i,...,b_j\}$ to an interval on the real line by the function g defined as follows:

$$g(b_i,...,b_j) = [f(b_i),f(b_j)] = [i,j].$$

Given a register conflict graph G for straight line code, there is an edge between two live ranges iff their intersection is not empty. Let G' be the graph derived from G by applying the function g to each node in G. Then G' is an interval graph, and a coloring for G' can be used to color G as well. Interval graphs can be colored optimally in polynomial time [Gavril 72] and hence an optimal global register allocation can be found in polynomial time for straight line code.

Figure 5-1 demonstrates how live ranges are mapped to intervals. Four live ranges, for variables a, b, c and d are shown. The live range for a consists of basics blocks $\{B1,B2,B3,B4,B5,B6,B7,B8\}$ and is mapped to the interval [1,8] etc. The real line is shown on to the left. The corresponding interval graph is shown to the right of the figure.

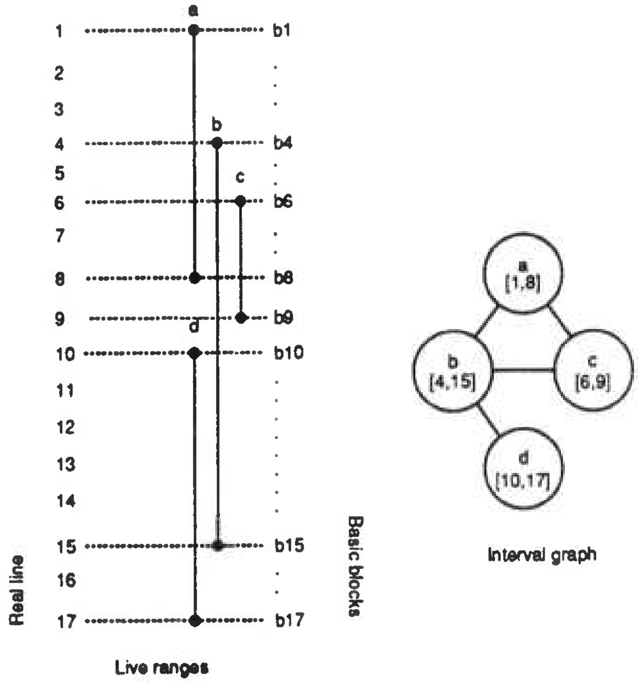


Figure 5-1: Register conflict graph for straight line code

One nice property of interval graphs is that it is easy to identify all the cliques they contain. In the next section we will show how all cliques in an interval graph are identified and how an order on all cliques is defined.

Definition 3: *(Clique)* A clique in a graph is a subset of vertices such that every pair of distinct vertices in that subset is connected by an edge.

It is easy to see that each node in a clique in a register conflict graph must be colored with a different color. The horizontal lines in the left part of Figure 5-1 are drawn at the end points of the intervals that correspond to the live ranges. Live ranges that intersect the same horizontal line form a clique in the corresponding interval graph. A new clique starts at each end point of an interval, shown as horizontal lines in Figure 5-1. In the following let n be the number of distinct live ranges. Then, the number of pairwise distinct end points of intervals is at most $2n$. This "sequence" of horizontal lines is the idea of ordering the cliques in an interval graph, which is formalized in the next paragraphs.

We first give a slightly modified definition for cliques that allows us to map cliques to the endpoints of the intervals corresponding to live ranges in the clique.

Definition 4: (*$[i,j]$clique*) We say that a set of intervals S forms a $[i,j]$ clique iff

1. $[i,j] \subseteq s \quad \forall s \in S$
2. $\exists s \in S$ such that $[i',j]$ is not a subset of $s \quad \forall i' < i$ or
3. $\exists s \in S$ such that $[i,j']$ is not a subset of $s \quad \forall j' > j$.

In other words, $[i,j]$ is the largest interval contained in every element of S.

In Figure 5-1, the interval [6,8] is contained in the intervals for live ranges a, b and c. Because the interval of live range a ends at 8, and the interval of live range c starts at 6, [6,8] is the maximal interval contained in all three live ranges. Hence, $\{a,b,c\}$ is a [6,8] clique. The intervals for a and b both contain [4,8]. Hence, $\{a,b\}$ is a [4,8] clique.

We can now define an ordering on the $[i,j]$ cliques of an interval graph as follows:

Definition 5: *(The .<. order for cliques)* Let $[t_1,t_2]$ and $[t_3,t_4]$ be two intervals and c_1 be a $[t_1,t_2]$ clique and c_2 be a $[t_3,t_4]$ clique. We say that c_1 .<. c_2 iff $t_1 < t_3$. If $t_1 = t_3$, we say that c_1 .=. c_2.

Example: In Figure 5-1, $\{a,b\}$ is a [4,8] clique, and $\{a,b,c\}$ is a [6,9] clique. Hence, $\{a,b\}$.<. $\{a,b,c\}$. The .<. order can be used to order sets of cliques, more formally:

Definition 6: *(Clique sequence)* Let $clique_1, clique_2, \ldots, clique_n$ be a list of i,j cliques of an interval graph. We say that this enumeration of cliques is a *sequence* of cliques iff $\forall k,l \in \{1,\ldots,n\} \;\; k \leq l \;\rightarrow clique_k .\leq. clique_l$.

We have seen that register interference graphs for straight line code are interval graphs, because each live range can be mapped to an interval on the real line.

We will now show how we can obtain a sequence of cliques (Definition 6) from the interval graph that denotes live range conflicts of straight line code. The idea of the construction of a clique sequence consists of several parts. First, we construct a set S' of ordered non-overlapping intervals from the start- and endpoints of the live ranges such that S' covers all intervals of live ranges. We call this an interval cover of the live ranges. For each interval $s \in S'$, we determine the set of live ranges that overlap with s, called *cliques(s)*. This set of live ranges must form a clique, and we show that each such clique is a $[t_1,t_2]$ clique, where $s = [t_1,t_2]$. We then show that since the elements of S' are ordered, that if a live range s_1 starts at an earlier basic block than a live range s_2 then $clique(s_1) .\leq. clique(s_2)$. Hence we have found a sequence of cliques that covers all the cliques in an interval graph. Now more formally:

Definition 7: *(Interval cover)* Let $\{[a_1,b_1],\ldots,[a_n,b_n]\}$ be an ordered set of intervals, i.e. $\forall i\in\{1,\ldots,n-1\}\ \ a_i\leq a_{i+1}$. The ordered set of intervals $\{[a'_1,b'_1],\ldots,[a'_m,b'_m]\}$ is called an *interval cover* of $\{[a_1,b_1],\ldots,[a_n,b_n]\}$ iff

1. $a'_1 = a_1$
2. $b'_m = b_n$
3. $\forall i\in\{1,\ldots,m-1\}\ \ b_i\ \ a_{i+1}-1$
4. $\forall j\in\{1,\ldots,m\}\ \exists\ i\in\{1,\ldots,n\}$ such that $a_j'=a_i\vee b_j'=b_i$

For an example, we turn again to Figure 5-1. The set of live ranges consists of four intervals: [1,8] = *a*, [4,15] = *b*, [6,9] = *c*, [10,17] = *d*. The interval cover of those four live ranges consists of intervals $\{[1,3][4,5][6,8][9,10][11,15][16,17]\}$. It is easy to see that for each basic block b_i that is part of a live range, $f(b_i)=i$ occurs in exactly one of the intervals of the interval cover. Hence, the intervals of an interval cover are non-overlapping.

Definition 8: *(S-overlap clique of an interval)* Given a set *S* of intervals and an interval $[i,j]$, the S-overlap clique of $[i,j]$ is defined as $\{s\in S\ \mid\ [i,j]\subseteq s\}$.

The overlap clique of interval [4,5] in Figure 5-1 consists of live ranges $\{a,b\}$.

Lemma 9: Given the set *S* of live ranges in a straight line program with interval cover $\{[a_1,b_1],\ldots,[a_m,b_m]\}$, $i<j$ implies that

$$(\text{S-overlap clique of } [a_i,b_i])\ .\!<\!.\ (\text{S-overlap clique of } [a_j,b_j])$$

Proof: By definition of interval cover.

Lemma 10: Given the set *S* of live ranges in a straight line program, let $\{[a_1,b_1],\ldots,[a_n,b_n]\}$ be the interval cover of *S* and let $clique_i$ be the *S*-overlap clique of interval $[a_i,b_i]$. Then $clique_i$ is a $[a_i,b_i]$ clique.

Proof: By definition of overlap cliques, the live ranges that are members of the overlap cliques must contain the interval $[a_i,b_i]$. Hence, all live ranges in that clique contain this interval, and the clique is thus a $[a_i,b_i]$ clique.

Given an interval graph *G*, we can construct a sequence of cliques by first constructing the interval cover for the nodes in *G*. The sequence of overlap cliques of the intervals in the interval cover is a sequence of cliques, as in Definition 6. It remains to be shown that this sequence of cliques is exhaustive, i.e. any clique in an interval graph is contained in that sequence of cliques.

Theorem 11: Let $\{[a_1,b_1],\ldots,[a_n,b_n]\}$ be the interval cover of the nodes of an interval graph *G*, and let $clique_1,\ldots,clique_n$ be the corresponding sequence of overlap cliques. For each clique *c* in *G* $\exists$ at least one $i\in\{1,\ldots,n\}$ such that *c* is contained in $clique_i$.

Proof: Given the members $m_1,\ldots,m_m$ of clique *c*, there must be at least one basic block b_j that is contained in each live range m_k, $k\in\{1,\ldots,m\}$, otherwise *c* would not be a clique. By construction of interval covers, there is exactly one interval $[a_p,b_p]\in\{[a_1,b_1],\ldots,[a_n,b_n]\}$ such that $b_j\in[a_p,b_p]$. By Lemma 10, $clique_p$ of interval $[a_p,b_p]$ is an $[a_p,b_p]$ clique, hence all live ranges containing b_j are in $clique_p$ - *q.e.d.*

Theorem 11 states that the cliques in an interval graph can be ordered to form a sequence of cliques. Therefore it is easy to determine the size of the largest clique in an interval graph, and thus *k* colorability can be determined in polynomial time. We will now show that the node removal technique colors any interval graph optimally. Since we know the clique sequence for an interval graph denoting register conflicts, it is easy to find the largest clique. If the largest clique is not larger than k, the node removal technique will produce an optimal coloring in linear time. This is stated in the next theorem and lemma.

Theorem 12: Given an interval graph G in which the size of the largest clique is k, there exist at least two nodes v_1 and v_2 in G that have at most $k\text{-}1$ neighbors in G.

Proof: Let $clique_1,...,clique_n$ be the clique sequence for G constructed from the interval cover of G. Let $clique_i$ be the first clique in that sequence such that there is a node v that is in $clique_{i\text{-}1}$ but not in $clique_i$. Hence, for $j \in \{2,...,i\}, clique_{j-1} \subset clique_j$. Because the size of the largest clique is k, $clique_i$ can be at most of size k. Because v can only be a member of the cliques $\{clique_1,...,clique_i\}$, $clique_i$ has "accumulated" all clique members of $\{clique_1,...,clique_{i\text{-}1}\}$, v can have at most $k\text{-}1$ neighbors.

Let $clique_j$.<. $clique_n$ be the last clique to which a new live range w is added, that is $\forall k \in \{j,..,n-1\}$ $clique_{k+1} \subset clique_k$. Hence, $clique_j$ is the largest clique in the subsequence $clique_j, clique_{j+1},...,clique_n$, and w is in each clique of that subsequence. Hence, w can not have more than $k\text{-}1$ neighbors.

Lemma 13: Let G be an interval graph in which the size of the largest clique is k. Then the node removal technique introduced in [Chaitin 81] will determine k colorability of G.

Proof: By Theorem 12, there must be at least two nodes v and w in G with less than k neighbors. Hence, both v and w can be removed. Let the graph G' derived from G by removing v and w. G' is an interval graph in which the largest clique has at most k members, and hence there are at least two more nodes in G' that are removable. G' is guaranteed to get smaller after each removal step, and eventually G' is the empty graph - *q.e.d.*

We have shown that the standard node removal technique will determine a k-coloring for an interval graph if there exists one. In general, the standard method fails to provide a k-coloring for graphs that are non-interval graphs. In the course of this chapter we will discuss the shape of register conflict graphs as they occur for complex programming constructs like loops and conditionals. We will see that register conflict graphs for loops and conditionals can be arbitrary graphs. Even if a k-coloring exists for such conflict graphs, the standard node removal technique might be unable to produce one. We will see that there are cases in which structural analysis of register conflict graphs enables us to find k-colorings that are not detected by the standard node removal technique. We first discuss aspects of register conflict graphs for loops.

5.2. Straight line loops and circular arc graphs

We first concentrate on register conflict graphs for loops that consist of loop entry *entry*, loop exit *exit* and a loop body b that consists of straight line code. We will show that the class of register conflict graphs for such simple loops contains arbitrary circular arc graphs, and that such graphs are therefore NP hard to color optimally.

Figure 5-2 shows a loop that consists of a sequence of definition and use statements. Each definition or use statement forms a basic block. The numbers of the basic blocks are shown to the right of each statement. Thus the loop shown in Figure 5-2 consists of loop entry 1, loop exit 13 and the straight line loop body {2,3,4,5,6,7,8,9,10,11,12}. The loop contains loop-continuous live ranges for b, c, d, e and f and of one loop-broken live range for a, namely {12,13,1,2,3,4,5,6}. Each loop-continuous live range of the loop can be mapped to an interval on the real line, since the loop body consists of straight line code. The live range for *b* consists of basic blocks {8,9,10,11} and hence *b* is mapped to the interval [8,11]. Because *a*'s live range is a loop-broken live range, it consists of basic blocks that are not contiguous. By definition of loop-broken live ranges, there is a definition of *a* whose next use can only be reached via the loop's backarc. Hence, *a*'s live range can not be mapped to a single interval. Instead, *a*'s live range consists of a set of intervals - one interval per contiguous set of basic blocks. In Figure 5-2, the live range for *a* ({12,13,1,2,3,4,5,6}) is mapped to two intervals, [12,13] and [1,6] respectively. The mapping of the live ranges in the loop to intervals is shown to the right of the loop construct. Note that there is a dotted line between the two intervals that represent the live range *a*. This means that both intervals represent the same live range. The dotted portion between the two parts of the live range for *a* corresponds to the hole in live range *a* consisting of {7,8,9,10,11}.

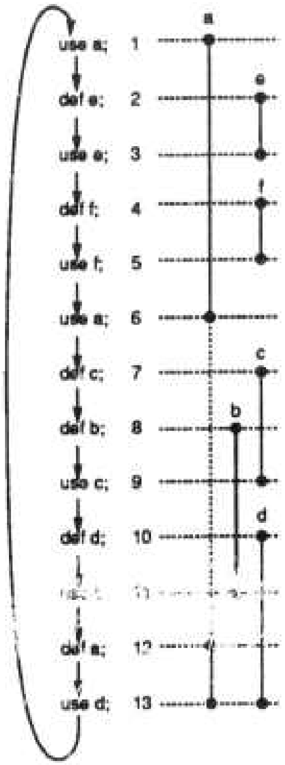


Figure 5-2: Register conflict graph for loop

A loop is usually executed many times, i.e. if the loop of Figure 5-2 were executed, the definition of *a* in block 12 would be followed by blocks 13, and then 1, hence the execution sequence of the basic blocks that form the live range for *a* is contiguous. Multiple executions of a loop can be expressed by a circle consisting of the set of basic blocks that form a straight line loop. This is depicted in Figure 5-3. The basic blocks that form the same loop as in Figure 5-2 are arranged along a circle. The execution order is clockwise, hence basic block 13 is followed by basic block 1 etc. Given this representation of a loop, the live ranges of the loop can be arranged around that circle. This is shown in Figure 5-4. The live ranges are segments of the circle formed by the basic blocks of the loop. Hence, the live range for *b* consists of the segment between basic blocks 8 and 11, and the live range for *a* is the (continuous) segment between 12 and 6.

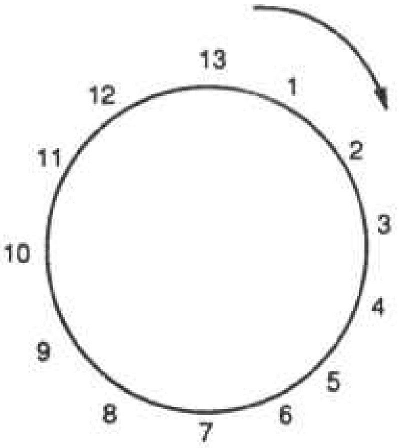


Figure 5-3: Loop expressed as a circle of basic blocks

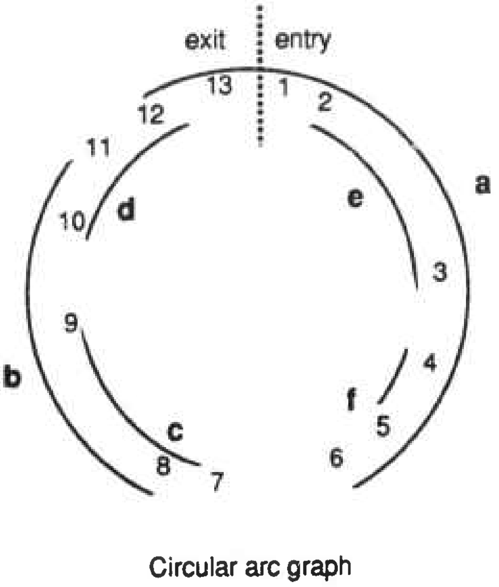


Figure 5-4: Register conflict graph for loop

> **Definition 14:** *(Circular arc graph)* A circular arc graph is a graph whose vertices can be represented as segments of a circle such that two vertices are adjacent if and only if the corresponding circle segments overlap.

It is easy to see that register conflict graphs for loops can be arbitrary circular arc graphs. Optimal coloring of an arbitrary circular arc graph is NP complete [Garey, M.R. and Johnson, D.S. 79]. Since for an arbitrary circular arc graph a loop can be constructed such that the live ranges of the loop arranged on a circle are equal to the vertices of the circular arc graphs, register conflict graphs for arbitrary loops are NP hard to color.

In the next paragraphs, we discuss situations in which the register conflict graph of a loop *L* is "independent" of the backarc in *L*.

5.3. Simplifying loops for the purpose of global register allocation

We have seen in the previous section that the presence of a loop-broken live range was necessary to construct a register conflict graph that is an arbitrary circular arc graph. At least one connected component of the live range graph of a loop-broken live range must contain the backarc of the loop by definition of loop-broken. Removing the backarc from a loop causes that connected component to be separated into two unconnected pieces - the corresponding register conflict graph changes. We now show situations in which the register conflict graph of a loop L is *unaltered* when the backarc is removed from L.

5.3.1. Removing the backarc in the absence of loop-broken live ranges

If all live ranges in a loop L are continuous, no live range can have any holes. The register conflict graph of the program L' derived from L by removing the backarc is identical to the register conflict graph of L. Given a loop L with a loop body that consists of complex programming constructs, the backarc of L can still be ignored, as long as all live ranges are *loop-continuous* in L. In other words, the backarc of a loop L can be ignored for the purpose of global register allocation if L does not cause any holes in live ranges. This is stated in the next Lemma.

Lemma 15: Given a loop L with loop head h and loop exit e that does not contain any loop-broken live ranges and in which all continuous equivalent live ranges have been changed into continuous live ranges by adding the appropriate instructions described in Lemma 11 in Chapter 4, let L' be the flow graph derived from L by removing the edge from node e to node h (the backarc). Then the register conflict graphs for L and L' are identical.

Proof: The set of live ranges in L' is equal to the set of live ranges in the original loop L. Otherwise, there would be a live range $l' = \{b_j,...,b_n\}$ in the derived loop L' but not in the original loop L. This is only possible if $\{b_1,...,b_j\}$ is part of a connected component in a variable v's live range graph in L that contains L's backarc, and removing the backarc causes a partition of that connected component. Let $\{b_1,...,b_{j-1},b_j,...,b_n\}$ be that connected component in v's live range graph in L. By definition of live range graphs, b_j must contain a definition of v that is not preceded by a use of v in b_j, and removing L's backarc disconnects $\{b_1,...,b_{j-1}\}$ and $\{b_j,...,b_n\}$. Then, there must be a basic block $b_u \in \{b_1,...,b_{j-1}\}$ that uses the definition in b_j such that b_u is part of L but not part of any conditional or loop nested inside L. By prerequisite all broken live ranges must contain a hole; if removing L's backarc causes a separation of a broken live range, the hole of live range $\{b_1,...,b_{j-1},b_j,...,b_n\}$ must be linked to L - a contradiction because by prerequisite all live ranges in L are loop continuous.

Because the set of live ranges in both L and L' is identical, the set of edges between the live ranges must be identical. Hence, the conflict graphs for L and L' are identical - *q.e.d.*

The example shown in Figure 5-5 illustrates Lemma 15. The flow graph labeled *F1* is a loop that contains two loop-continuous live ranges, one for a and one for b respectively. The flow graph labeled *F2* is derived from *F1* by removing the backarc. The conflict graph for *F1* is equal to the conflict graph for *F2*, and is shown to the right.

As a consequence of Lemma 15, the register conflict graph for a program that contains no conditionals and in which all live ranges are continuous is equivalent to the register conflict graph of straight line code.

We have seen how the absence of loop-broken live ranges in a loop L enables us to simplify the flow

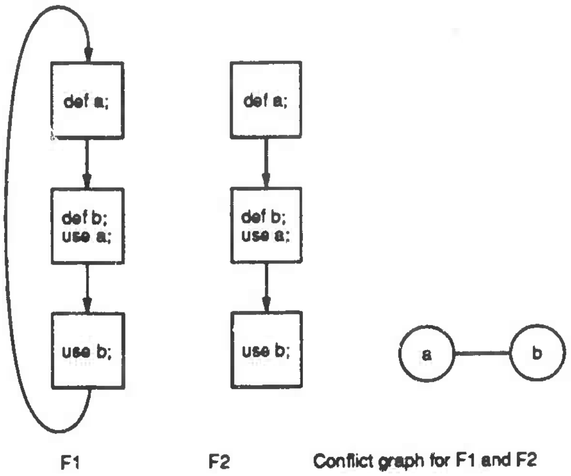


Figure 5-5: Register conflict graph for a loop with forward live ranges

graph by removing the backarc of *L* for the purpose of global register allocation. If there is a loop-broken live range in a loop *L*, removing *L*'s backarc in general results in a different register conflict graph, depicted in Figure 5-6. The loop depicted to the left has a loop-broken live range for *a* consisting of basic

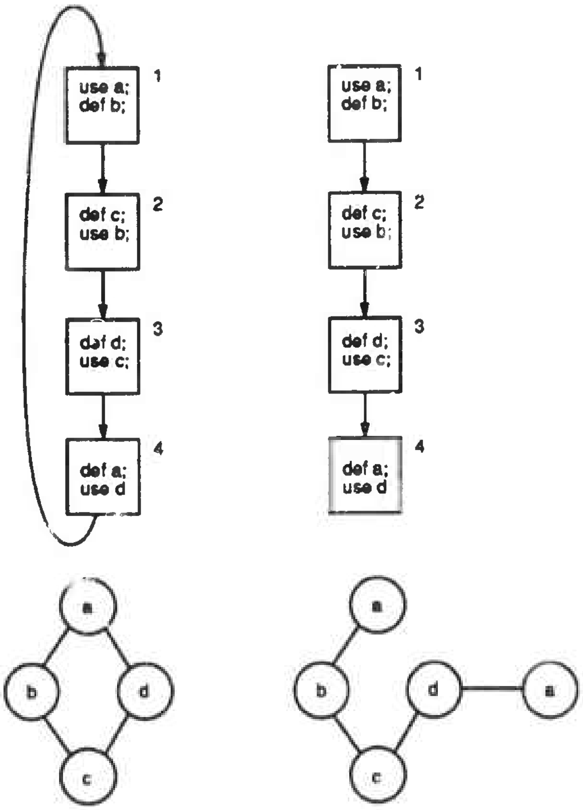


Figure 5-6: Removing the backarc in the presence of loop-broken live ranges

blocks {1,4}. The register conflict graph for that loop is depicted below. Removing the backarc partitions the live range for *a* into two separate live ranges, {1} and {4} respectively. This leads to an extra node for *a* in the register conflict graph - one per live range. The register conflict graph for the flow graph to the right is again shown below and differs from the register conflict graph of the original loop. In the next section, we discuss restrictions that must be met if the register conflict graph of a loop L is independent of L's backarc in the presence of loop-broken live ranges.

5.3.2. Removing the backarc in the presence of loop-broken live ranges

Given a loop L with a loop-broken live range v, v must contain a hole that is entirely contained in L. Note that by definition of loop broken, the hole can not be entirely contained in any programming construct nested inside L. We can therefore partition a loop-broken live range into two parts. The first part of the live range consists of basic blocks in the top part of the loop and contains the loop entry, and the second part of the live range consists of basic blocks in the bottom part of the loop and contains the loop exit. In the following, we call those parts *TOP* and *BOT*. Before we give a formal definition of the *TOP* and *BOT* parts of a loop-broken live range we demonstrate the concept with a few examples.

Figure 5-7 depicts a sample loop with one loop-broken live range for variable a, consisting of $\{1,2,7,8\}$. The live range has one hole consisting of $\{3,4,5,6\}$. The live range is partitioned into the top part *TOP* = $\{1,2\}$ containing the loop entry and the bottom part *BOT* = $\{7,8\}$ containing the loop exit. A more complex example is shown in Figure 5-8. A complex loop is shown, consisting of loop head *B0*, loop exit *B5* and a loop body that consists of a conditional. In that loop, a is a loop-broken live range; the *BOT* part of a consists of $\{B2,B4,B5\}$ and the *TOP* part of a consists of $\{B0\}$. Note that the definitions of a in basic blocks *B2* and *B4* occur in different branch clauses. Therefore the basic blocks that form the *BOT* part of the live range for a do not form straight line code. The *BOT* part of the live range for a consists of the connected component of the live range graph for a that contains the loop exit. We are now ready to define the *TOP* and *BOT* part of a loop-broken live range formally.

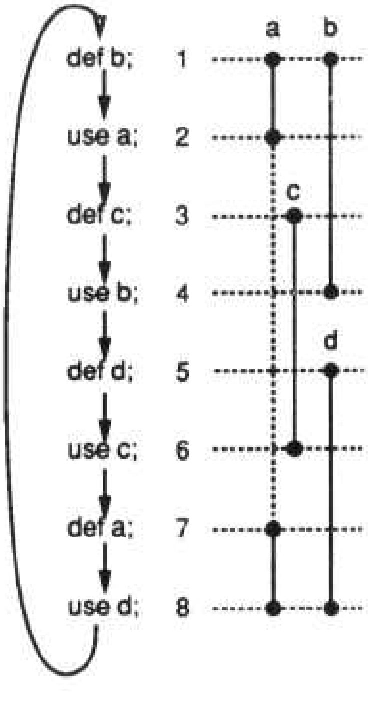


Figure 5-7: *TOP* and *BOT* part of a loop-broken live range

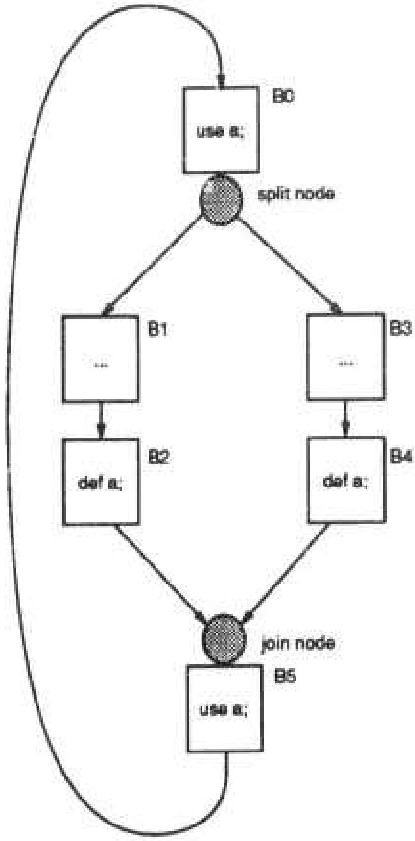


Figure 5-8: *TOP* and *BOT* part of a loop-broken live range in a complex loop

Definition 16: *(TOP and BOT set of a loop-broken live range)* Given a loop-broken live range for a variable v of a loop L with loop head h and loop exit e and the live range graph for v, $TOP(v,L)$ consists of the basic blocks that form the connected component of v's live range graph that contains h. $BOT(v,L)$ consists of the basic blocks that form the connected component of the live range graph that contains e.

In the next paragraphs we will discuss situations that permit to eliminate the basic blocks that form the *TOP* part or the *BOT* part from a loop broken live range without changing the register conflict graph.

Definition 17: *(Definition starting the BOT part of a loop-broken live range)* Given a loop-broken live range for a variable x in a loop L, we say that a definition of x in basic block b_x starts $BOT(x,L)$ iff $b_x \in BOT(x,L)$.

Example: in the loop depicted in Figure 5-8, basic blocks *B2* and *B4* both contain a definition of a and are members of the set $BOT(a,L)$. Hence, both definitions start the *BOT* part of the live range for a.

Given a loop L in which no loop-continuous live range contains a basic block that occurs in $BOT(y,L)$ of a loop-broken live range y, it is easy to see that in the register conflict graph the "bottom part" of y can be ignored in the loop, illustrated in Figure 5-9. In the loop depicted to the left, loop-broken live range a consists of *TOP* part {1,2} and *BOT* part {6,7}. Neither loop-continuous live range b nor c overlap with a in basic blocks 7 or 8. The *BOT* part consisting of basic blocks 7 and 8 can be eliminated by removing a's definition in basic block 7 and the subsequent uses in the loop that are reachable on paths that do not

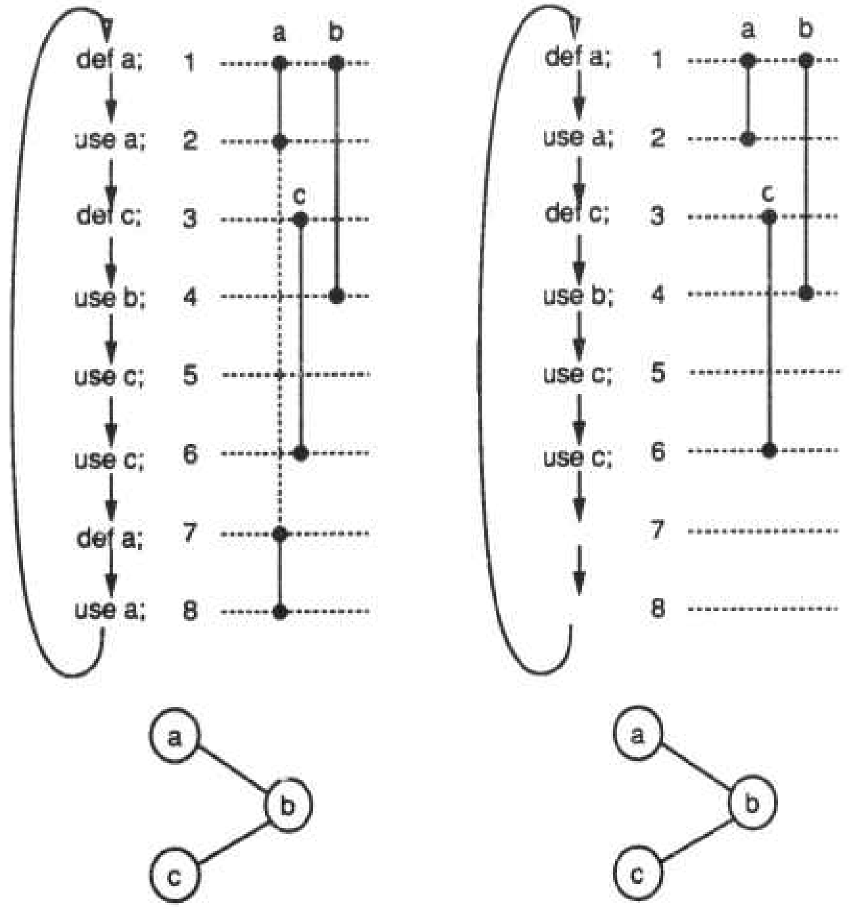


Figure 5-9: Removing the bottom part of a loop-broken live range

contain the loop's backarc. This is depicted to the right. For both loops the register conflict graphs are shown below - they are identical.

Note that the removal of the definition that starts the *BOT* part of *a*'s live range does not alter the register conflict graph even if there is a loop-continuous live range that contains basic blocks of *both* the *TOP* part and the *BOT* part of *a*. This is demonstrated in Figure 5-10. loop-continuous live range *b* contains basic blocks from both the *TOP* part and the *BOT* part of *a*. Still the register conflict graph does not change when the definition of *a* in basic block 7 is removed.

Given a loop *L*, removing definitions and uses that form the *BOT* part of a loop-broken live range *l* is equivalent to eliminating the entire *BOT* part of *l*. If we enforce that every live range that overlaps with the *TOP*(*l*,*L*) of a loop-broken live range *l* also overlaps with *BOT*(*l*,*L*), then the entire *BOT* part of *l* can be eliminated from *l* without altering the register conflict graph. These observations are formalized in the next lemma.

> **Lemma 18:** Given a flow graph *F* with register conflict graph *G* that contains a loop *L* consisting of basic blocks $\{b_1,...,b_n\}$ with loop-broken live ranges $l_1,... l_n$, let l_j' be derived from l_j by removing all basic blocks that are in $BOT(l_j,L)$ $\forall$ $j \in \{1,...,m\}$, and let *G'* be derived from *G* by exchanging l_j' for $l_j \forall j \in \{1,...,m\}$. If every live range that contains a basic block of the *BOT*

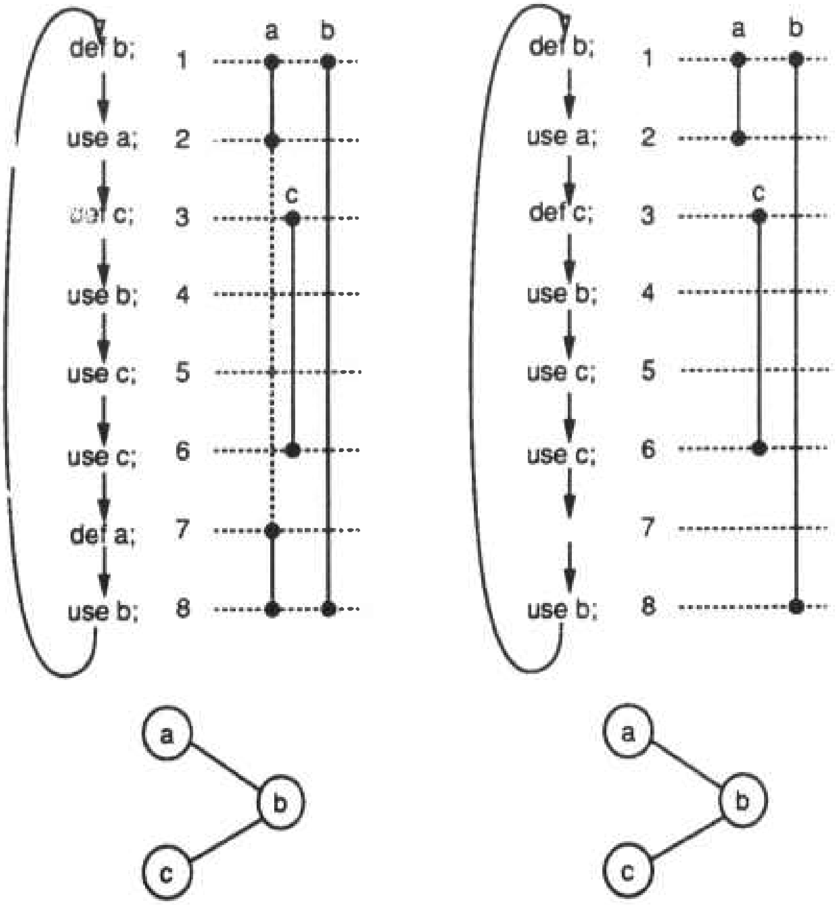


Figure 5-10: loop-continuous live range overlapping with both the *BOT* and *TOP* set of a loop-broken live range

set of a loop-broken live range also contains at least one basic block of the *TOP* set of the same live range, *G* and *G'* are identical.

Proof: The nodes in *G* consist of nodes representing loop-broken live ranges and of nodes representing loop-continuous live ranges. Every loop-continuous live range *f* that co-exists with a loop-broken live range *l* contains either just basic blocks that occur in members of *TOP(l,L)* or both basic blocks that occur in members of *TOP(l,L) and* basic blocks that occur in members of *BOT(l,L)*. Therefore the node representing *f* exists both in *G* and *G'*. Further, the set of edges incident to the node for *f* must be identical in both *G* and *G'* because if *f* co-exists with *l* in a basic block occurring in a member of *BOT(l,L)*, it must also co-exist with *l* in a basic block occurring in a member of *TOP(l,L)* and therefore there must be an edge between *f* and *l* in *G'*. Since all other live ranges of *L* are unaltered, *G* and *G'* are equal - *q.e.d.*

Note that Lemma 18 also holds if the terms *BOT* and *TOP* are exchanged. This is illustrated in Figure 5-11. The live ranges of a loop are depicted to the left; *a* is a loop-broken live range with *TOP* part {1,2} and *BOT* part {7,8}. Live range *b* is the only loop-continuous live range adjacent to *a*, and it shares with the live range for *a* only basic blocks that are in the *BOT* part of *a*. Removing the uses of *a* in basic blocks 1 and 2 eliminates the *TOP* part of *a* - the register conflict graph for both loops is identical and shown at the bottom of the figure.

Lemma 19: Given a flow graph *F* with register conflict graph *G* that contains a loop *L* consisting of basic blocks $\{b_1,...,b_n\}$ with loop-broken live ranges $l_1,...,l_n$, let l_j' be derived from l_j by removing all basic blocks that are in $TOP(l_j,L)$ ∀ $j \in \{1,...,m\}$, and let *G'* be derived from *G*

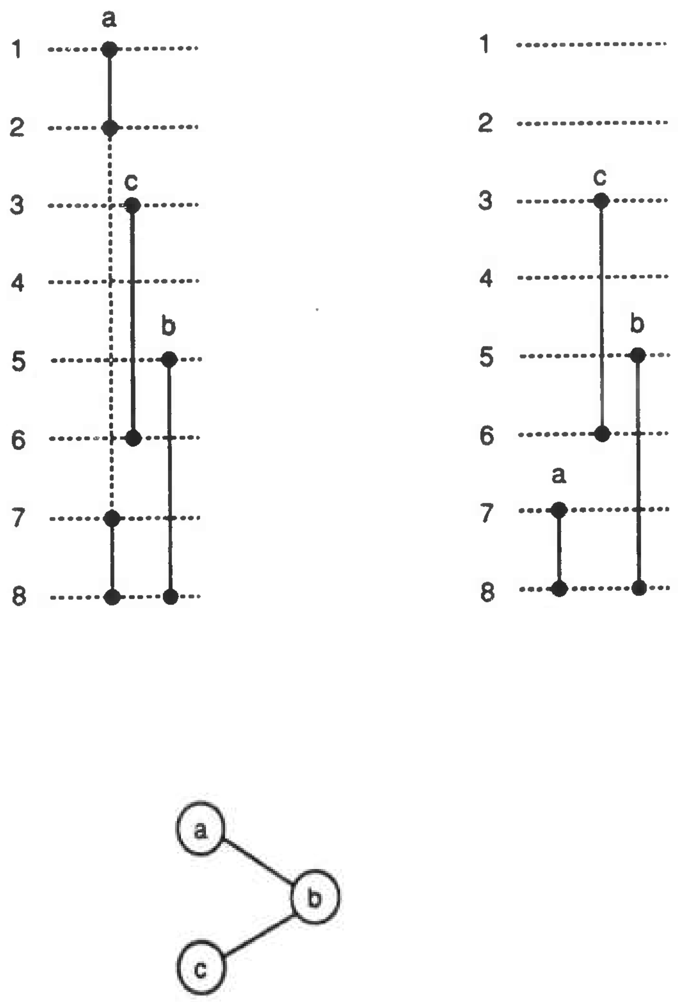


Figure 5-11: Removing the *TOP* part of a loop-broken live range

by exchanging l_j' for $l_j \forall j \in \{1,...,m\}$. If every live range that contains a basic block of the *TOP* set of a loop-broken live range also contains at least one basic block of the *BOT* set of the same live range, *G* and *G'* are identical.

Proof: The nodes in *G* consist of nodes representing loop-broken live ranges and of nodes representing loop-continuous live ranges. Every loop-continuous live range *f* that co-exists with a loop-broken live range *l* contains either just basic blocks that occur in members of *BOT(l,L)* or both basic blocks that occur in members of *BOT(l,L) and* basic blocks that occur in members of *TOP(l,L)*. Therefore the node representing *f* exists both in *G* and *G'*. Further, the set of edges incident to the node for *f* must be identical in both *G* and *G'* because if *f* co-exists with *l* in a basic block occurring in a member of *TOP(l,L)*, it must also co-exist with *l* in a basic block occurring in a member of *BOT(l,L)* and therefore there must be an edge between *f* and *l* in *G'*. Since all other live ranges of *L* are unaltered, *G* and *G'* are equal - *q.e.d.*

Note that the removal of basic blocks described in Lemmas 18 or 19 turn loop-broken live ranges into loop-continuous live ranges. If all loop-broken live ranges can be changed to loop-continuous live ranges, the backarc of the loop can be removed without changing the register conflict graph of the loop (Lemma 15). By definition, both the *TOP* and the *BOT* part of a loop broken live range *l* in a loop *L* can contain basic blocks outside the loop *L*. Because we enforce that *every* live range that overlaps with *TOP(l,L)* also overlaps with *BOT(l,L)*, we can remove the loop backarc without changing the register conflict graph of the *entire* program.

All that was discussed in section 5.3.2 can be applied to conditional-broken live ranges: if all live ranges that overlap with a conditional-broken live range b overlap with both the *TOP* and the *BOT* set of b, the overall register conflict graph does not change when the *TOP* or the *BOT* part of b are removed. Hence, the conditional-broken live range b has been changed into a conditional-continuous live range.

We have seen that in the absence of loop-broken live ranges, the backarc of a loop can be removed without altering the register conflict graph. In other words, the flow graph that previously contained a loop has been "straightened out" and now no longer contains that loop. The simplification of loops depends merely on the absence of loop-broken live ranges. For conditionals, the absence of broken live ranges is not enough to ensure a register conflict graph that is easy to color optimally. We will see that even if every live range in a conditional is continuous, register conflict graphs for conditionals can be arbitrary graphs that are hard to color optimally.

5.4. Register conflict graphs for conditionals

Throughout this section we assume that no loop that is part of a branch clause contains loop-broken live ranges. Examining register conflict graphs for conditionals is more complex than for loops and straight line code, because there is a larger variety of live range types, and there are alternative *paths* through the branch.

> **Definition 20:** *(Path through a conditional branch)* A path through a conditional branch with split node s and join node j is a sequence of basic blocks $\{b_1,...,b_n\}$ such that $b_1=s$, $b_n=j$, $\forall i \in \{1,...,n\}$, b_i occurs in a branch clause and $\forall i \in \{2,...,n\}$ $<b_{i-1},b_i>$ is an edge in the flow graph denoting the branch.

Examples for paths through the conditional branch depicted in Figure 5-12 are the sequences $\{B0 \rightarrow B1 \rightarrow B3 \rightarrow B4 \rightarrow B8\}$ and $\{B0 \rightarrow B5 \rightarrow B6 \rightarrow B7 \rightarrow B8\}$, etc.

Each path through a conditional branch consists of straight line code. While the register conflict graph for one individual path can be colored optimally in isolation, it is in general not possible to combine optimal colorings for individual paths to an optimal coloring for the entire conditional statement in polynomial time. An exception are conditional branches in which each live range is *BLOCAL*, discussed next.

5.4.1. Conditionals in which all live ranges are *BLOCAL*

Intuitively, the number of *BGLOBAL*, *LOEN* and *LOEX* live ranges in a conditional is a measure for the degree of dependency between the register conflict graphs of the individual branch clauses. The reason is that *BGLOBAL*, *LOEN* and *LOEX* live ranges might be shared between the register conflict graphs of separate branch clauses. If each live range in a conditional is *BLOCAL*, the register conflict graphs for individual branch clauses are completely independent. Therefore, conditionals in which all live ranges are *BLOCAL* can be colored optimally by combining individual colorings for each branch clause. This is formalized in the next theorem.

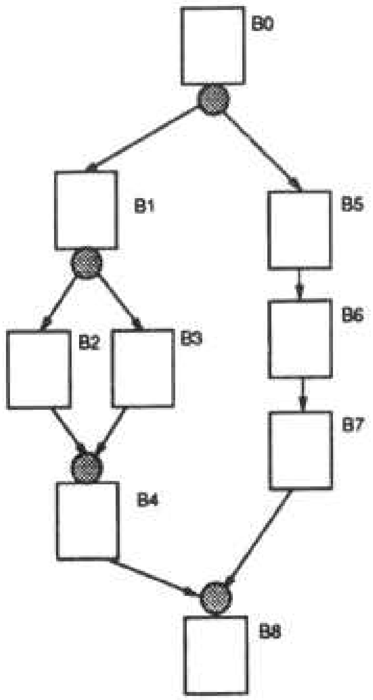


Figure 5-12: Paths through a conditional

Theorem 21: Given a register conflict graph G of a conditional branch with split node s and join node j, and a set of branch clauses $c_1,...,c_n$, the chromatic number of G is equal to the maximal chromatic number of $G_{c_i}, i \in \{1,...,n\}$ if all live ranges that occur in the conditional branch are *BLOCAL*.

Proof: The conflict graph G for the entire conditional can be partitioned into the conflict graph for the split node G_s, the conflict graph for the join node G_j, and a conflict graph for each branch clause G_{C_i}. Both G_s and G_j must be empty, because the conditional branch contains only *BLOCAL* live ranges and no *BLOCAL* live range can contain s or j. For the same reason, there can be no edge between the conflict graphs for the individual branch clauses. Hence, $G=\cup(G_{C_i}, i \in \{1,...,n\})$ and the chromatic number of G is equal to the maximal chromatic number of the G_{C_i} - *q.e.d.*

Note that Theorem 21 holds for nested conditionals, even if the inner conditionals do contain live ranges that are not *BLOCAL*. As a consequence of Theorem 21, the register conflict graph of a conditional C with split node S, join node J, and branch clauses $c_1,...,c_n$ in which all live ranges are *BLOCAL* is equivalent to the register conflict graph of the flow graph derived from C by "linearizing" the branch clauses. Because the register conflict graphs for the clauses are completely unrelated, the order in which the branch clauses are linearized in the flow graph is irrelevant.

5.4.2. Conditionals and *BGLOBAL* live ranges

The presence of *BGLOBAL* live ranges in a conditional complicates things, because *BGLOBAL* live ranges can be conditional-broken live ranges. A *BGLOBAL* live range for a variable v is conditional-broken if there is a re-definition of v in some branch clause. An example of a continuous and a broken *BGLOBAL* live range is given in Figure 5-13. The live range for variable a depicted to the left consists of

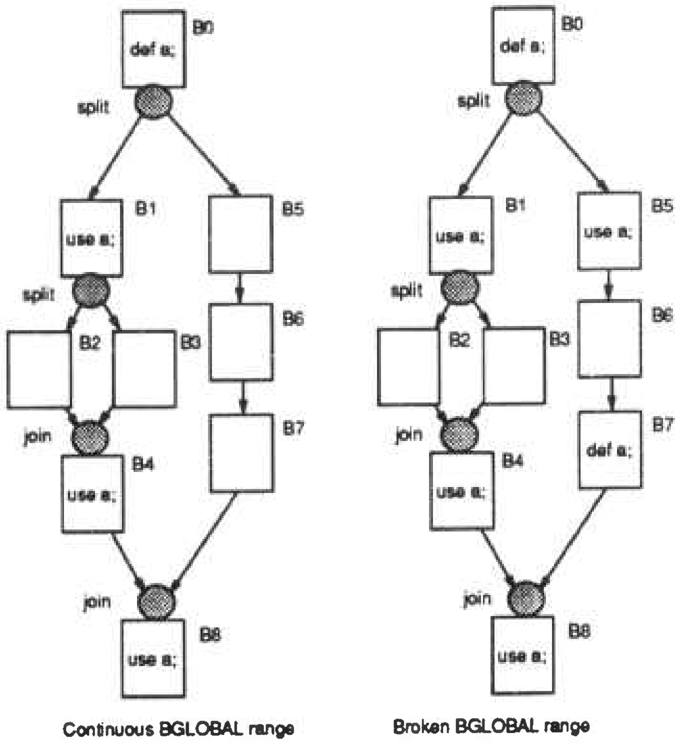


Figure 5-13: Continuous and broken *BGLOBAL* live ranges

basic blocks $\{B0,B1,B2,B3,B4,B5,B6,B7,B8\}$, that is all basic blocks of the conditional branch. Hence, the live range is a continuous *BGLOBAL* live range. The live range for *a* depicted to the right in figure 5-13 consists of basic blocks $\{B0,B1,B2,B3,B4,B5,B7,B8\}$. *B6* is not part of the live range for *a*, even though there exists a path through the branch that contains *B6*. Hence, it is a broken *BGLOBAL* live range.

One more reason why *BGLOBAL* live ranges can make the register conflict graph of a conditional more complex is because they introduce dependencies between the register conflict graphs of individual branch clauses. In the next paragraphs we will see how the continuity of *BGLOBAL* live ranges determines how hard it is to color register conflict graphs for conditionals that contain *BLOCAL* and *BGLOBAL* live ranges.

5.4.2.1. Continuous *BGLOBAL* live ranges in conflict graphs for conditionals

A continuous *BGLOBAL* live range for a variable *v* consists of all basic blocks that are on a path from the split node to the join node of a branch. Therefore, a continuous *BGLOBAL* live range overlaps with *every* other live range in the conditional. In the register conflict graph for a conditional branch, a node that represents a continuous *BGLOBAL* live range is therefore adjacent to all other nodes.

Lemma 22: Given the register conflict graph G of a conditional that contains n continuous *BGLOBAL* live ranges, let G' be the graph derived from G by removing all n nodes that represent continuous *BGLOBAL* live ranges. The chromatic number of G is equal to the chromatic number of $G' + n$.

Proof: Let c' be the chromatic number of G', and let each $\{c_1,...,c_m\}$ be the set of colors used for the nodes in G'. Each continuous *BGLOBAL* live range is adjacent to each vertex in G' and hence must be colored with a color $c \notin \{c_1,...,c_m\}$. Since the number of *BGLOBAL* live ranges is n, the chromatic number of G is therefore $c' + n$.

As a consequence of Lemma 22 continuous *BGLOBAL* live ranges can be "ignored" during the coloring process, because they require each a new color, regardless of the specific coloring found for the other nodes in the conflict graph.

If all *BGLOBAL* live ranges are *conditional-continuous*, the branch clauses of a conditional can be linearized for the purpose of register allocation if the conditional does not contain any *LOEN* or *LOEX* live ranges. The reason is that all holes in *BGLOBAL* live ranges are contained inside an inner loop or conditional and thus inside a branch clause. Note that conditional continuity is a weaker condition than continuity.

Broken *BGLOBAL* live ranges can make register conflict graphs for conditionals arbitrarily hard to color: like loop-broken live ranges, conditional-broken live ranges can cause arbitrary circular arc register conflict graphs, discussed in the next paragraphs.

5.4.2.2. Conditionals and broken *BGLOBAL* live ranges

We show now that conditionals that consist of *BLOCAL* and arbitrary *BGLOBAL* live ranges produce register conflict graphs that include arbitrary circular arc graphs, and are therefore NP hard to color.

Before we give a formal theorem, we go through an example that illustrates the point. Figure 5-14 shows a conditional that contains a broken *BGLOBAL* live range for a.

The conditional consists of two branch clauses, one consisting of the single block 14, the other consisting of a straight line segment {1,2,3,4,5,6,7,8,9,10,11,12,13}. The conditional contains one broken *BGLOBAL* live range for variable a, consisting of basic blocks {0,1,2,3,4,5,6,12,13,14,15}. The live range for a is broken, because it does not contain all nodes along the path through the branch clause that starts with block 1.

Because each path through a conditional consists of straight line code, the live ranges of path {0,1,2,3,4,5,6,7,8,9,10,11,12,13,15} can be mapped to intervals on the real line like live ranges of straight line code. This is depicted to the right in Figure 5-14. The live range for a is mapped to two intervals, [0,6] and [12,15], because a is not live in all basic blocks in the path. Because a's live range is *BGLOBAL*, the intervals [0,6] and [12,15] must be colored with the same color, indicated by a dotted arc between them. Note that this register conflict graph is equivalent to that depicted for a loop in Figure 5-2. Hence, the class of register conflict graphs for conditionals contains circular arc graphs. In the next theorem we show that for an arbitrary circular arc graph, there is a conditional branch whose register conflict graph is equal to the circular arc graph.

Theorem 23: Given an arbitrary circular arc graph C, there exists a conditional branch B that contains only *BLOCAL* and broken *BGLOBAL* live ranges such that the register conflict graph for B is equal to C.

Proof: Let $\{[a_1,b_1],....,[a_n,b_n]\}$ be the segments of a circle. Suppose the circle is labeled with the endpoints of the segments $\{c_1,...,c_m\}$ such that $\forall i \in \{1,..,m\} \exists j \in \{1,...,n\}$ such that $c_i = a_j$ or $c_i = b_j$ and $\forall i \in \{1,...,m-1\}\, c_i < c_{i+1}$.

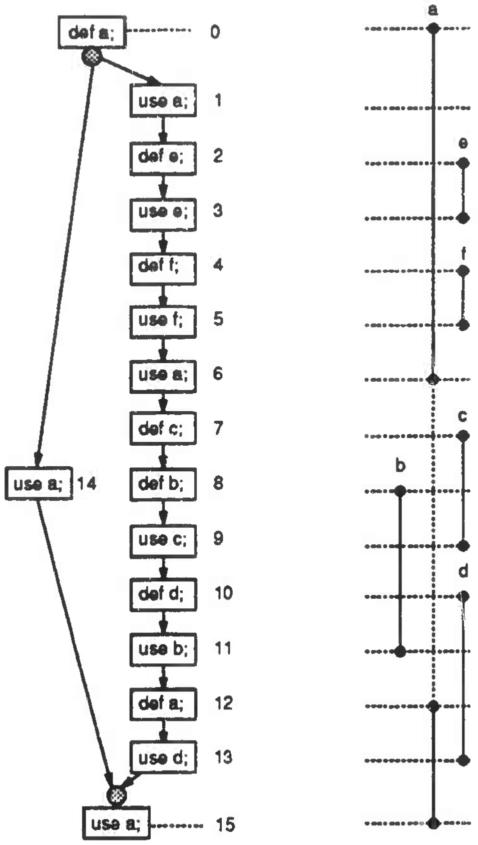


Figure 5-14: Mapping live ranges of one path to intervals

We construct a conditional with split node c_0, join node c_{m+1} and two branch clauses. The newly introduced c_0 and c_{m+1} are chosen such that $c_0 < c_1$ and $c_{m+1} > c_m$.

The first branch clause is empty, i.e. there is an edge from the split node to the join node. The second branch clause consists of the straight line code segment $\{c_1,...,c_m\}$.

Each circle segment in $\{[a_1,b_1],....,[a_n,b_n]\}$ is mapped to a set of intervals on the real line as follows:

$$f([a_i,b_i])=\begin{cases}[a_i,b_i] & \text{if } a_i \le b_i\\ [a_i,c_{m+1}],[c_0,b_i] & \text{if } a_i \ge b_i\end{cases}$$

The live ranges in our constructed branch are the intervals $f([a_i,b_i])$. If $a_i \ge b_i$, the circle segment $[a_i,b_i]$ is mapped to two intervals, and both intervals are part of the same live range. Because the split node c_0 and the join node c_{m+1} are part of that live range, it must be a *BGLOBAL* live range. It is easy to see that the register conflict graph for this branch is equal to the original circular arc graph - *q.e.d.*

The proof of 23 is best understood by an example. Figure 5-15 depicts a circular arc graph, with segments a=[1,4],b=[3,5],c=[6,10],d=[7,8] and e=[9,2]. We construct a conditional B such that the register conflict graph for B is equal to the circular arc graph in Figure 5-15.

The circle is labeled $\{1,2,3,4,5,6,7,8,9,10\}$, i.e. $c_1 = 1$ and $c_m = 10$. We choose $c_0 = 0$ and $c_{m+1} = 11$. The results of the function f defined above are as follows:

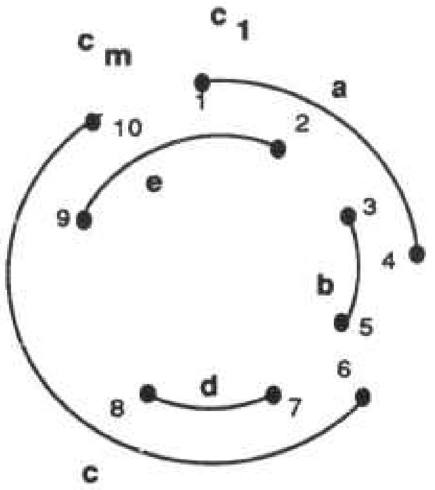


Figure 5-15: An arbitrary circular arc graph

(a)	$f([1,4])=[1,4]$
(b)	$f([3,5])=[3,5]$
(c)	$f([6,10])=[6,10]$
(d)	$f([7,8])=[7,8]$
(e)	$f([9,2])=([9,11],[0,2])$

We construct a conditional that consists of branch split node $c_0=0$, join node $c_{m+1}=11$ and one empty branch clause, and a straight line branch clause consisting of basic blocks $\{1,2,3,4,5,6,7,8,9,10\}$. The live range for *e* is *BGLOBAL*, hence *e* is live at the split node and at the join node. All other live ranges are *BLOCAL*. Figure 5-16 shows the conditional whose register conflict graph is the circular arc graph depicted in Figure 5-15.

Note that *e* is a broken *BGLOBAL* live range - by Lemma 22 and Theorem 21 register conflict graphs of conditionals that contain just *BLOCAL* live ranges and continuous *BGLOBAL* live ranges can be colored optimally in polynomial time and hence can not be arbitrary circular arc graphs.

In the presence of *LOEN* and *LOEX* live ranges, register conflict graphs for conditionals can become very complicated. We will see that even if *all* live ranges in a conditional are continuous, the presence of *LOEN* and *LOEX* live ranges can cause an arbitrary register conflict graph. There is no easy solution to why *LOEN* and *LOEX* live ranges cause arbitrary graphs - we therefore restrict ourselves to discussing situations in which *LOEN* and *LOEX* live ranges do *not* cause arbitrary register conflict graphs. For the remainder of this chapter, we assume that all live ranges in a conditional are continuous, so that we can concentrate on the influence of *LOEN* and *LOEX* live ranges.

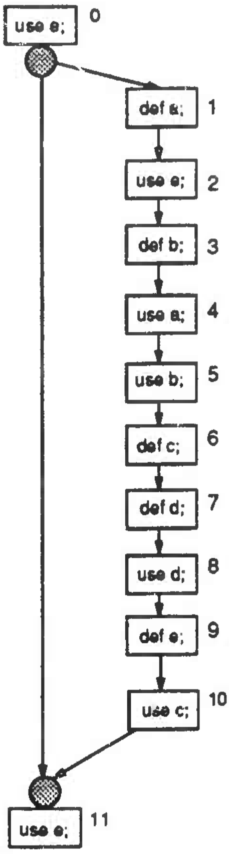


Figure 5-16: Branch constructed from an arbitrary circular arc graph

5.4.3. Conditionals and *LOEN*/*LOEX* live ranges

By definition of *LOEN* live ranges, all *LOEN* live ranges of a conditional contain the split node and therefore co-exist during program execution. At the same time, all *LOEX* live ranges contain the join node. Hence, all *LOEN* live ranges must reside in distinct registers and all *LOEX* live ranges must reside in distinct registers.

It is easy to see that if *all LOEN* live ranges overlap with *all LOEX* live ranges, the corresponding nodes must form a clique in the register conflict graph. By definition, *LOEN* and *LOEX* live ranges must all overlap with *BGLOBAL* live ranges. If in addition a conditional *C* contains no *BLOCAL* live ranges, each *LOEN* and *LOEX* live range can be changed into a continuous *BGLOBAL* live range without changing the register conflict graph. This is done as follows: all *LOEN* and *LOEX* live ranges are "padded" until they contain all basic blocks that form *C*. Thus, the register conflict graphs for the individual branch clauses share all nodes - the register conflict graph of an arbitrary branch clause is equal to the register conflict graph of the entire conditional. Therefore, all but one branch clause can be eliminated from the flow graph for the purpose of register allocation - the conditional can be "collapsed" into one arbitrary branch clause.

If the *LOEN* and *LOEX* live ranges of a conditional do not form a clique in the register conflict graph, colors used for a *LOEN* live range *a* can be re-used for a *LOEX* live range *b* as long as there is no basic block in which *a* and *b* co-exist.

Definition 24: *(Overlap set of a LOEN node)* Given a conditional C, the *overlap set* of *LOEN* live range a in C is the set of *LOEX* live ranges $x_1,x_2,...,x_n$ in C such that there is at least one basic block in which a and x_i, $i \in \{1,...,n\}$ co-exist.

All *LOEX* live ranges in a live range a's overlap set must be colored with colors that differ from a. At the same time *LOEN* colors can be be re-used for *LOEX* live ranges not in the overlap set.

Figure 5-17 shows the register conflict graph of a conditional that consists only of *LOEN* and *LOEX* live ranges. In that example, a,b,c and d are *LOEN* live ranges. The set of *LOEX* live ranges consists of v,w,x,y and z. Live range a overlaps with *LOEX* live ranges w,x and z; b overlaps with y and v etc. By definition of *LOEN*, the *LOEN* live ranges form a clique; the same is true for the *LOEX* live ranges. If a maximal number of colors used to color the *LOEN* clique can be re-used to color the *LOEX* clique, the coloring for that conflict graph is optimal. The problem of re-using the maximal number of colors in the *LOEX* clique can be mapped to the maximal matching problem for a bipartite graph.

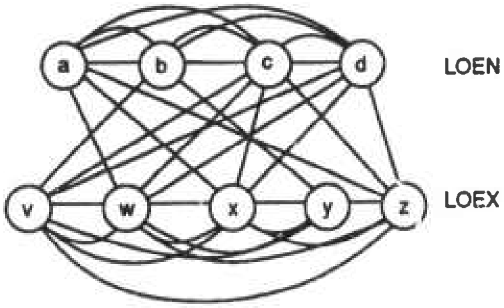


Figure 5-17: Register conflict graph consisting of *LOEN/LOEX* live ranges

Definition 25: *(Matching in a graph)* A matching in a graph G is an independent subset of its edges, such that no two of the edges are adjacent. A *maximal* matching of G is the largest possible independent set of edges.

We construct a bipartite graph from the set of *LOEN* live ranges and *LOEX* live ranges as follows:

1. each live range corresponds to one unique node,
2. there is an edge between a *LOEN* live range a and each *LOEX* live range that is not in a's overlap set.

Figure 5-18 depicts the bipartite graph derived from the register conflict graph shown in Figure 5-17.

LOEN live range a overlaps with *LOEX* live ranges w,x,z. Therefore the *LOEX* nodes adjacent to a are v,y. It is easily seen that the graph is bipartite. A maximal matching of the *LOEN* nodes with *LOEX* live ranges can be used to re-use colors of *LOEN* life ranges for *LOEX* live ranges safely. Finding an optimal matching for the nodes of a bipartite graph can be done in polynomial time. An algorithm for optimal bipartite matching can be found in [Smith 87].

The maximal matching of the derived bipartite graph is used to derive a coloring for the conflict graph as follows: An arbitrary color is chosen for each *LOEN* live range and for each unmatched *LOEX* live range. Matched nodes receive the color of their "matching"-partner. The detailed algorithm is given in Figure 5-19.

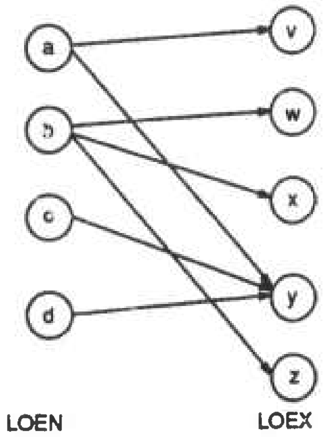


Figure 5-18: Bipartite graph consisting of *LOEN* and *LOEX* live ranges

```
Input:          The LOEN and LOEX and BGLOBAL live ranges of a non nested branch
Output:         An optimal coloring for those live ranges
Method:

        for all LOEN live ranges l do
                determine the overlap set;
                color each l with a distinct color;
        od

        G := the empty graph;

        for all LOEN live ranges l do
                for all LOEX live ranges x ∈ overlap(l) do
                        add x and l G's vertices if not yet in it;
                        add edge <l,x> to G's edge set;
                od
        od

        bipartite matching(G);

        for all LOEX live ranges x do
                if x has been matched
                then color x with the matched nodes color
                else color x with a new color;
        od
```

Figure 5-19: Coloring a register conflict graph for a conditional that contains only *LOEN* and *LOEX* live ranges

The restriction on the live ranges in a conditional that allows an opimal coloring can be relaxed by allowing arbitrary *BGLOBAL* live ranges in addition to *LOEN* and *LOEX* live ranges. The idea behind this relaxation is that *BGLOBAL* live ranges by definition contain *both* the split node *and* the join node, regardless of whether they are continuous or broken. Hence, every *BGLOBAL* live range co-exists with every *LOEN* live range in the split node, and with every *LOEX* live range in the join node. Therefore, the registers assigned to *BGLOBAL* live ranges in a conditional must differ from the registers assigned to *LOEN* and *LOEX* live ranges. This is all summarized in the next theorem.

Theorem 26: Given a conditional that contains only *LOEN*, *LOEX* and *BGLOBAL* live ranges and the corresponding register conflict graph *G*, an optimal coloring for *G* can be found in polynomial time.

Proof: A coloring for the *LOEN* and *LOEX* live ranges can be found by algorithm 5-19. After application of algorithm 5-19, a new color is chosen for every *BGLOBAL* live range, and the overall coloring is optimal - *q.e.d.*

Note that the bipartite matching algorithm can no longer be used to find an optimal coloring in the presence of *BLOCAL* live ranges because in each branch clause *BLOCALS* can be adjacent to an *arbitrary* set of *LOEN* and *LOEX* live ranges. It is not possible to incorporate dependencies of *BLOCALS* into the bipartite graph. We will see in the next section that under certain restrictions for *LOEN* and *LOEX* live ranges it is possible to color register conflict graphs for conditional branches optimally in polynomial time, even if a conditional contains a combination of *BLOCAL* and *LOEN* and *LOEX* live ranges.

5.4.4. Mixing *LOEN* and *LOEX* live ranges with *BLOCAL* live ranges

We have seen that register conflict graphs for straight line code are interval graphs. Interval graphs are contained in the class of *chordal* graphs.

Definition 27: *(Chord)* Given a cycle in a graph $\{c_1,...,c_n\}$, a chord in the cycle is an edge between two non-consecutive members of the cycle.

An example of a cycle and chords in the cycle is shown in Figure 5-20. The cycle consists of nodes {1,2,3,4,5,6,7,8}; the chords are the dashed edges *<2,7>* and *<1,5>*.

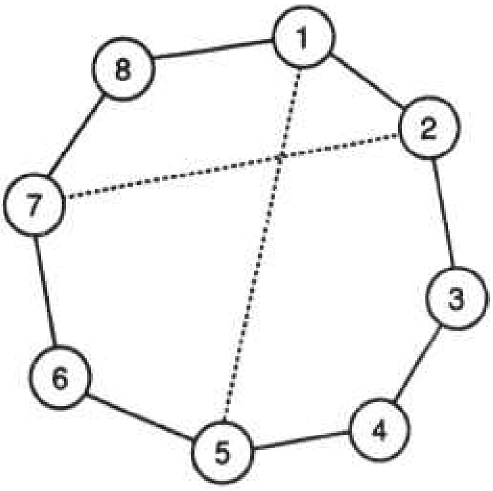


Figure 5-20: Chords in a cycle

Chordal graphs are graphs that are "triangularizable", more formally:

Definition 28: *(Chordal graph)* A graph *G* is chordal iff every cycle in *G* that consists of more than 3 nodes has a chord.

Graphs *G1* and *G2* depicted in Figure 5-21 are chordal because *G1* and *G2* contain no chordless cycle larger than 4. Note that graph *G1* consists of a collection of "triangles". Graph *G3* is derived from *G1* by adding an edge between nodes 7 and 2, and is no longer chordal because cycles {1,2,7,8} and {2,3,4,5,6,7}

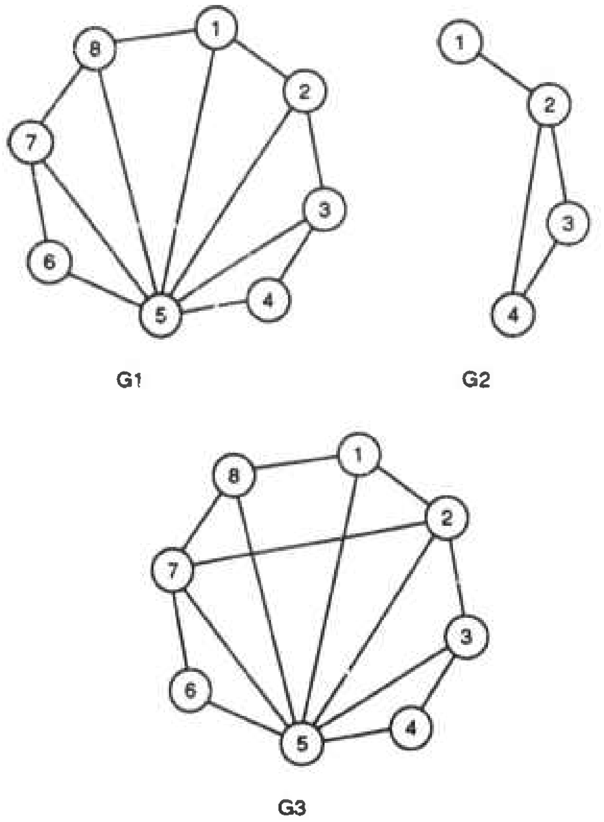


Figure 5-21: Examples of chordal and nonchordal graphs

are chordless. A nice property of chordal graphs is they can be colored optimally in polynomial time [Gavril 72].

We will show that under certain restrictions for *LOEN*, *BGLOBAL* and *LOEX* live ranges, register conflict graphs for conditionals are chordal. Before we formalize the restrictions in a theorem, we give some intuitive reasons for chordality of register conflict graphs.

Given a conditional branch with *n* branch clauses, let G_{c_i} be the register conflict graph for the *i*-th branch clause. It is easy to see that if the register conflict graphs of two distinct branch clauses have nodes in common, those nodes must be *BGLOBAL*, *LOEN* or *LOEX* live ranges. This is illustrated in Figure 5-22.

We see a conditional construct consisting of split and join node and two branch clauses. The first branch clause consists of basic blocks 1,2 and 3, the second branch clause consists of basic blocks 4,5,6 and 7. The branch contains live ranges for 6 variables. Since each variable has one unique live range, we name the live ranges after the variables. Live range *n* is a *LOEN* live range, and live range *x* is *LOEX*. The left branch clause contains *BLOCAL* live range *d*, the right branch clause contains *BLOCAL* live ranges *a*,*b* and *c*.

Figure 5-23 shows the register conflict graphs for the first branch clause, labeled *G1*, the second branch clause, labeled *G2* and the register conflict graph of the entire conditional branch.

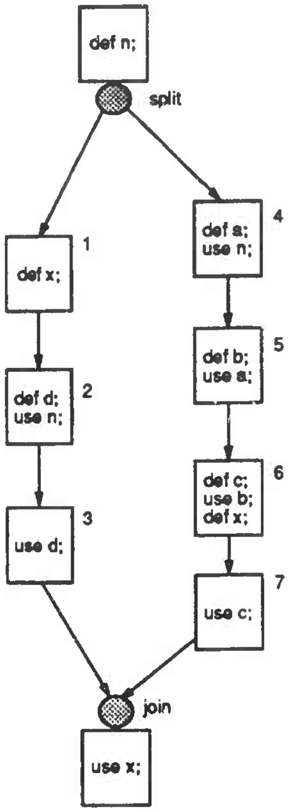


Figure 5-22: Conditional branch

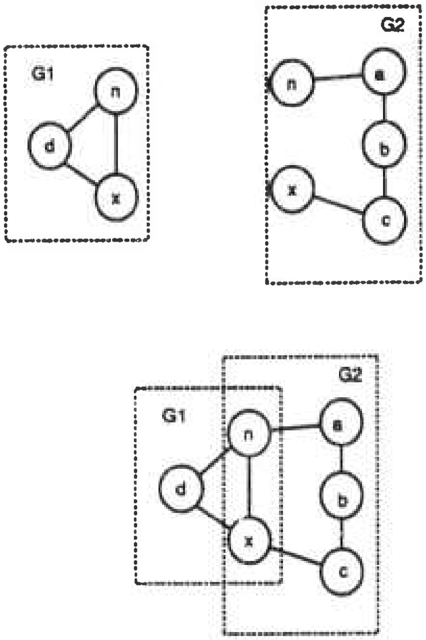


Figure 5-23: Nonchordal register conflict graphs

Note that live ranges *x* and *n* occur in both *G1* and *G2*, but that in the first branch clause *x* and *n* co-exist: this is not the case in the second branch clause. Hence, in *G1* there is an edge between *n* and *x*, but not in *G2*. The register conflict graph of the entire branch construct contains an edge between *n* and *x*, and the graph contains a chordless cycle consisting of nodes $\{a,b,c,n,x\}$.

Figure 5-24 shows the same conditional branch construct as in Figure 5-22, only now the live ranges *n* and *x* overlap in *both* the first branch clause and the second branch clause.

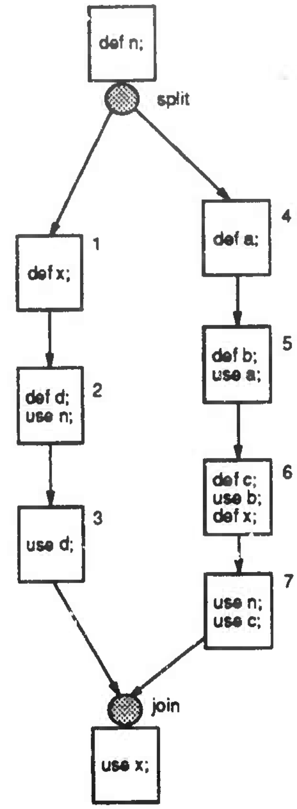


Figure 5-24: Conditional branch construct with chordal register conflict graph

This changes both the register conflict graph for the second branch clause, labeled *G2* and the register conflict graph for the entire conditional branch as shown in Figure 5-25.

Note that the register conflict graph for the entire branch construct contains no longer a chordless cycle that is larger than 3. In other words, the register conflict graph is chordal. We formalize the observations about chordality of register conflict graphs for conditionals in a sequence of lemmas.

We now analyze types of live ranges that can be part of chordless cycles larger than 4. First, we show that there must be at least one pair of non-overlapping *LOEN/LOEX* live ranges if a chordless cycle of size larger than 4 consists only of *BGLOBAL* nodes.

> **Lemma 29:** Given a register conflict graph *G* of a non-nested conditional that contains no broken live ranges, let $\{c_1,\ldots,c_n\}$ be a chordless cycle in *G* of 4 or more nodes, such that c_i is either a *BGLOBAL*, a *LOEN* or a *LOEX* live range for $i \in \{1,\ldots,n\}$. Then there must be at least one *LOEN* live range c_i and one *LOEX* live range c_j such that c_j is *not* in c_i's overlap set.
>
> **Proof:** Suppose that the overlap set of every *LOEN* live range $\in \{c_1,\ldots,c_n\}$ includes every *LOEX* live range $\in \{c_1,\ldots,c_n\}$. Then there is an edge between every *LOEN* live range $\in \{c_1,\ldots,c_n\}$ and every *LOEX* live range in $\in \{c_1,\ldots,c_n\}$. Further, the *LOEN* live ranges form a clique as do the *LOEX* live ranges. By definition, all *BGLOBAL* live ranges overlap with every member in the cycle. Hence, the nodes in $\{c_1,\ldots,c_n\}$ form a clique and therefore cannot form a chordless cycle that is larger than 3. Hence, there must be at least one *LOEN* live range c_i and one *LOEX* life range c_j such that c_j is not in c_i's overlap set - *q.e.d.*

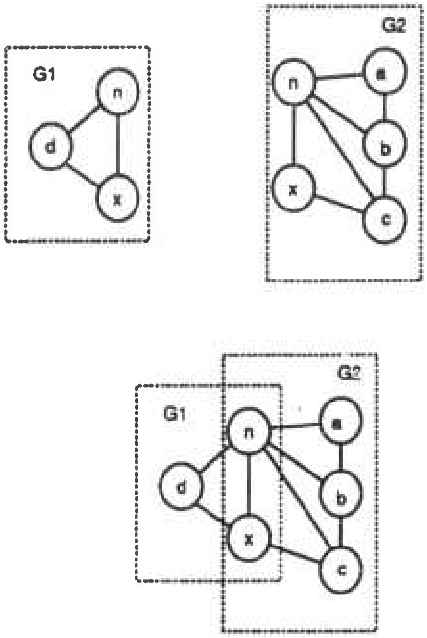


Figure 5-25: Chordal register conflict graphs

A chordless cycle of size 4 or larger can not consist of live ranges that are *BLOCAL*, stated in the next lemma.

Lemma 30: Given a register conflict graph G of a non-nested conditional that contains no broken live ranges, let $\{c_1,...,c_n\}$ be a chordless cycle in G of 4 or more nodes. Then there must be at least two nodes in the cycle that are not *BLOCAL* in every branch.

Proof: Suppose that all nodes in $\{c_1,...,c_n\}$ are *BLOCAL*. By definition, *BLOCAL* live ranges contain neither a split node nor a join node. Consequently, any *BLOCAL* live ranges that are connected must be part of straight line code that contains no split or join nodes. Register conflict graphs of straight line code are interval graphs and hence can not contain a chordless cycle that is larger than 3.

There must be at least *two* nodes in the cycle that are not *BLOCAL*, for suppose there is only *one* node c_g in $\{c_1,...,c_n\}$ that is not *BLOCAL*. Because there exist no edges between *BLOCAL* nodes in different branch clauses, there must be a chordless cycle of size 4 or larger that consists of c_g and nodes that are *BLOCAL* and all part of the same branch clause. This is not possible, because the subgraph formed by c_g and *BLOCAL* live ranges of one branch clause is an interval graph and can not contain a chordless cycle larger than 3 - *q.e.d.*

Via Lemmas 29 and 30 we can now show that every chordless cycle in a register conflict graph of a non-nested conditional branch must contain at least one pair *LOEN/LOEX* live ranges x and y for which there is a branch clause in which x and y do not co-exist.

Theorem 31: Given a register conflict graph G of a non-nested conditional branch that contains no broken *BGLOBAL* live ranges, let $\{c_1,...,c_n\}$ be a chordless cycle in G of 4 or more nodes that contains *BLOCAL* live ranges. Then there must be at least one *LOEN* live range x and one *LOEX* live range y such that there is at least one branch clause in which x and y do not co-exist.

Proof: By Lemma 30 there must be at least two members of $\{c_1,...,c_n\}$ that are not *BLOCAL*. Let x and y be two arbitrary members of the cycle that are not *BLOCAL*.

If x and y are both continuous *BGLOBAL* live ranges, they co-exist in every branch clause by definition. Hence, x and y must be either *LOEN* or *LOEX*.

We will now assume that *all LOEN* and *LOEX* live range overlap in every branch clause. Then there can be at most two *LOEN* or *LOEX* live ranges in the cycle, because by our assumption more than two *LOEN* or *LOEX* live ranges must contribute a chord to the cycle. Hence the cycle contains $\{x,y,c_1,...,c_{n-2}\}$ and all $c_i \in \{c_1,...,c_{n-2}\}$ must be *BLOCAL* live ranges. This is depicted in Figure 5-26. Live ranges x and y are non-*BLOCAL*, and live ranges *1,...,6* are *BLOCAL*.

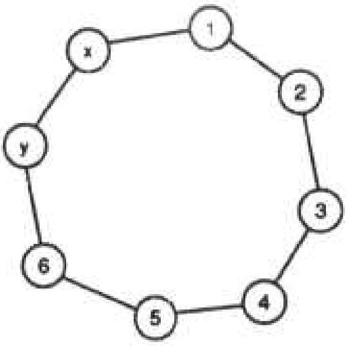


Figure 5-26: Chordless cycle with local live ranges 1,2,3,4,5,6

Because there are no edges between *BLOCAL* live ranges of distinct branch clauses, all the c_i in the cycle must be part of the same branch clause. By our assumption, x and y overlap in every branch clause. Because no $c_i \in \{c_1,..,c_{n-2}\}$ contains a split or join node, all $c_i \in \{c_1,..,c_{n-2}\}$ must occur in straight line code. But then the cycle can not be larger than 3.

It remains to be shown that x and y cannot be both *LOEN* or both *LOEX*. We will show that for the *LOEN* case - the *LOEX* case is similar.

Assume the cycle looks as above (depicted in Figure 5-26), and that both x and y are *LOEN* live ranges. We have shown that all c_i must be in the same branch clause and that x and y can not overlap in the branch clause in which live ranges $c_1,...,c_{n-2}$ occur. By definition, if x and y do not overlap in the branch clause containing $c_1,...,c_{n-2}$, either x or y can not be adjacent to any of $c_{i,i\in\{1,...,n-2\}}$. But there can only be a cycle if both x and y are adjacent to members of $\{c_1,...,c_{n-2}\}$ in the same branch clause - a contradiction. Hence, x and y can not both be *LOEN*. By the same arguments, x and y can not both be *LOEX*. As a result, x and y must be a *LOEN/LOEX* pair - *q.e.d.*

Theorem 31 states that if there exists a chordless cycle that is larger than *3* in a register conflict graph for a non-nested conditional branch that contains no broken *BGLOBAL* live ranges, there must be at least one pair of *LOEN* and *LOEX* live ranges x and y such that there is a branch clause in which x and y do not co-exist. It is easy to check if *LOEN* and *LOEX* live ranges co-exist in every branch clause, and if they all do, we can show that the register conflict graphs of such branches are chordal, stated in the following Lemma:

Lemma 32: Let G denote the register conflict graph of a non-nested conditional that contains no broken live ranges and in which all *LOEN* variables and all *LOEX* variables co-exist in every branch clause. Then G is a chordal graph and an optimal coloring for G can be found in polynomial time.

Proof: Since all *LOEN* and *LOEX* live ranges co-exist in every branch clause, the register conflict graph can not contain a chordless cycle larger than 3. Therefore, G is a chordal graph. See [Gavril 72] for an algorithm that produces an optimal coloring for chordal graphs *q.e.d.*

5.5. Chapter summary

We have started with the fact that register conflict graphs for straight line code are interval graphs, for which an optimal coloring can be found in polynomial time. We showed how the cliques in an interval graph can be ordered to form a ".<." sequence of cliques, and used the construction to prove that the standard node removal technique can color interval graphs optimally. The significance of finding the sequence of cliques in an interval graph is that each one of those cliques can be used as a separator clique.

We examined the register conflict graphs for loops and conditionals. Register conflict graphs for loops with arbitrary live ranges contain the class of circular arc graphs, and are thus NP hard to color optimally. If a loop consists only of continuous backarc live ranges, the register conflict graph of the loop is equivalent to the register conflict graph to the flow graph derived from the loop by removing the backarc live range.

Register conflict graphs for conditionals are more complex. We have seen that optimal colorings can be found for conflict graphs of conditionals with the following properties:

1. The live ranges in the conditional are either continuous *BGLOBAL* live ranges or *BLOCAL* live ranges w.r.t. each branch that contains them.
2. The conditional contains no *BLOCAL* live ranges.
3. Every pair of *LOEN* live range x and *LOEX* live range y co-exist in every branch clause and there are no broken *BGLOBAL* live ranges.

The motivation for examining situations in which loops can be "linearized" by removing the loop backarc and conditionals can be "linearized" or collapsed by ignoring or linearizing individual branch clauses is that these flow graph simplifications turn a complex flow graph into a "straightened out" flow graph that contains more straight line code sequences. Hence, more sequences more separator cliques can be found in the register conflict graphs. Each of those simplifications can only be carried out when the live ranges of the loop/conditional are "well behaved" and form a nice register conflict graph.

When these simplifications can not be carried out, it is possible that the register conflict graph is not an interval graph. In the next chapter we will discuss a technique called "node merging" with which a non-interval register conflict graph can be changed into an interval register conflict graph in some cases.

Chapter 6

Transformations on register conflict graphs

In the previous chapter we described situations that allow to simplify the flow graph for the purpose of register allocation without altering the register conflict graph. The flow graph can be simplified by removing backarcs from loops and sequentializing or collapsing conditionals. The purpose of transformations on flow graphs is to detect portions of the register conflict graphs that are interval graphs or chordal graphs. If those portions are known, it is easy to partition program flow graphs into clique connected components - the basis for parallelizing global register allocation.

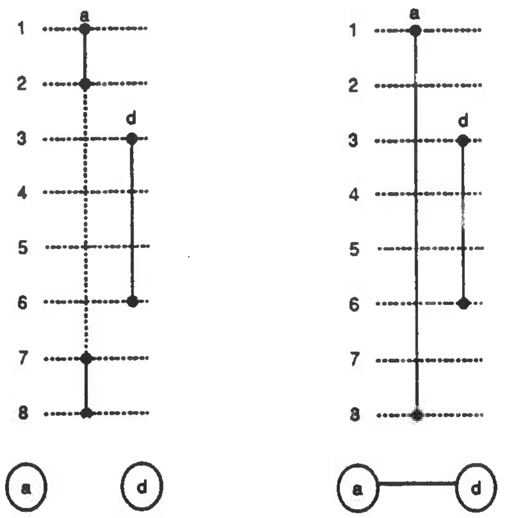
In this chapter, we deal with loops and conditionals that can not be simplified to equivalent straight line code. For such programs, register conflict graphs are usually not interval graphs. Our approach to register allocation is based on mapping non-interval register conflict graphs to interval register conflict graphs. Interval register conflict graphs are desirable for two reasons. First, they can be colored optimally in polynomial time. Second, clique separators are easy to locate in interval graphs. Clique separated portions of a register conflict graph can be colored individually and combined to an overall coloring by renaming only.

In our model, there are two sources of non-interval register conflict graphs: broken live ranges and unrestricted combinations of *LOEN* and *LOEX* live ranges in conditionals. If holes in live ranges can be "stuffed" with live ranges that "fit" into the holes, a broken live range can be transformed into a continuous one. We demonstrate situations in which broken live ranges can be transformed into continuous live ranges by a technique we call node merging. Node merging can also be applied to the nodes of the register conflict graphs of different branch clauses of the same conditional: given a conditional C with branch clauses c and c', it sometimes is possible to encode the register conflict graph of branch clause c into the register conflict graph of a different branch clause c'. A coloring of the register conflict graph for c' can be mapped directly onto the register conflict graph for c - hence, during register allocation it suffices to color only the graph for c'. If the conflict graphs for all branch clauses can be "reduced" to that of one single branch clause, the resulting conflict graph is much simpler and in some cases can even be colored optimally in polynomial time.

We first show how node merging can be applied to eliminate holes from broken live ranges. The second part of this chapter is devoted to the application of node merging to simplify register conflict graphs of conditionals.

6.1. The effect of holes in broken live ranges

Eliminating holes from broken live ranges means eliminating one source for non-interval register conflict graphs. The easiest way to eliminate a hole h from a broken live range v is to add all the blocks that form h to v. There are cases in which this increases the chromatic number of the register conflict graph unnecessarily, an example of which is given in Figure 6-1. The left side of the figure shows a broken and a continuous live range. Live range a consists of basic blocks {1,2,7,8} and contains a hole consisting of {3,4,5,6}, live range d consists of basic blocks {3,4,5,6}. The register conflict graph is shown to the right - there is no edge between the nodes for a and d and the conflict graph is 1-colorable. If the live range of a is padded to contain the basic blocks that form the hole {3,4,5,6}, the node for d can no longer be colored with the same color as the node for a - padding a causes an edge between nodes a and d, depicted on the right side of Figure 6-1 - the resulting conflict graph is no longer 1-colorable. The chromatic number of the register conflict graph increases because d is a live range that consists only of basic blocks that "fit" into the hole of live range a. Hence, padding a with the basic blocks that form the hole excludes the color used for a from the colors that can be used for d. Had there been no live range that fits into a's hole, padding a with the blocks that form the hole would have not altered the register conflict graph. This is formalized in the next lemma.



Original conflict graph

Conflict graph after padding

Figure 6-1: Padding a hole of a broken live range

Lemma 1: Given a flow graph with basic blocks B, live ranges L and register conflict graph G, let l be a broken live range with hole h. If there is no live range range v such that v consists of a subset of the basic blocks that form h, then adding h to l does not change G.

Proof: Let G' be derived from G by adding h to l. We prove by contradiction that G and G' must be equal, so we assume that G and G' differ. Because the set of nodes in G is equal to the set of nodes in G' except for l, the set of edges not incident on l must be equal as well. Clearly, the number of edges incident on l in G must be less than or equal than the number of edges incident on $l'=l \cup h$ because l consists of a subset of l'. Suppose there is a node v such that there is an edge between v and l' in G' but no edge between v and l in G. Then, v must contain basic blocks that occur in h. By prerequisite, v must then also contain basic blocks that occur in l, so there must be an edge between v and l in G - a contradiction.

As a consequence of Lemma 1, only a hole h for which there exists live ranges that "fits" into h can cause a non-interval register conflict graph. An example of a such non-interval register conflict graph is given in Figure 6-2. The live range for a is broken and contains hole {3,4,5,6}. The live range for c consists of

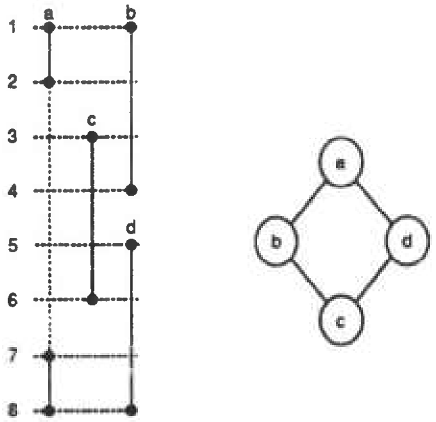


Figure 6-2: A broken live range and a non-interval register conflict graph

basic blocks {3,4,5,6}, and hence is a fit for the hole of the live range for a. The register conflict graph is shown to the right - it consists of a chordless cycle of size 4 and is therefore not an interval graph.

We will show that certain types of live ranges that fit into a hole of a broken live range can be used for padding that hole without penalty with regard to the chromatic number of the conflict graph. The definition of such live ranges is given below.

> **Definition 2:** *(Fit for a hole)* Given a flow graph consisting of basic blocks B with register conflict graph G, let l be a broken live range such that h is a hole of l. We say that a live range p is a fit for h iff p is continuous and $p\cap(B-h)=\emptyset$.

Figure 6-3 shows broken live range a with hole $h=\{3,4,5,6\}$ and a few examples of live ranges, some of which fit into h. The live range b is a fit for h, because b is continuous and consists of basic blocks that are

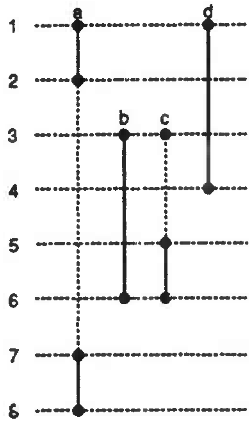


Figure 6-3: Fits for a hole

all contained in h. Because c is not a continuous live range, it is no fit for h, even though the basic blocks that form c, {3,5,6} are a subset of the basic blocks that form the hole. Because d contains basic blocks outside the hole, d is not a fit for h.

6.2. Eliminating holes via node merging

Node merging is a technique to enforce that two non-adjacent nodes in the register conflict graph are colored identically *before* an actual coloring is produced. Given a conflict graph G with two nodes v_1 and v_2, v_1 and v_2 are merged by eliminating node v_1 from G and adding all edges incident on v_1 to the edges incident on v_2. Under certain restrictions we can ensure that node merging changes a non-interval register conflict graph into an interval graph.

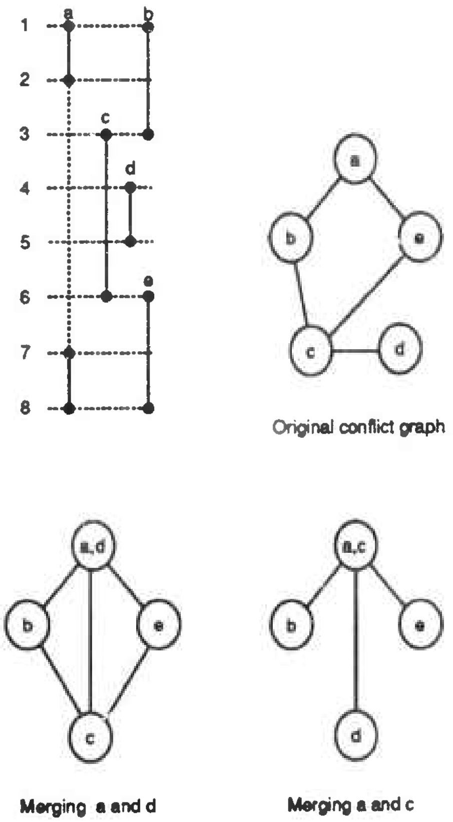
Node merging can be used to eliminate holes from broken live ranges. Given a broken live range l with a hole h and another live range p that is a fit for the hole h, there can not be an edge between l and p. Hence, it is possible to color l and p with identical colors. Chosing the same color for two separate live ranges can be viewed as "melting" the two live ranges into one - the live range l is changed into l' by adding the basic blocks contained in p. If padding l with p results in continuity of l', there is one less broken live range to worry about. We will discuss cases in which there are several fits for the same hole or in which several merge operations are necessary to eliminate one hole. We use node merging only if we can ensure that the chromatic number of G' derived from the original conflict graph G by live range merging is not larger than the chromatic number of G. Note that a coloring for the conflict graph derived by node merging can be directly mapped to the original conflict graph. In the next paragraphs we introduce restrictions that must be met by live ranges that fit into a hole such that the chromatic number of the conflict graph derived by node merging is equal to the chromatic number of the original conflict graph.

6.2.1. Perfect matches for holes

Figure 6-4 gives an example of a conflict graph that contains a broken live range a and two fits for its hole, c and d respectively. The original non-interval register conflict graph is shown top right. The register conflict graph derived by merging a and d is depicted bottom left. It is an interval graph, and contains two 3-cliques. Hence, 3 colors are needed to color the register conflict graph when d is merged with a. Merging a with c yields the interval graph depicted bottom right. It is easy to see that 2 colors are sufficient to color that graph. This example illustrates that deciding which live range to use for merging with a broken live range can be nontrivial.

In some cases the choice of the candidate for merging with a broken live range is irrelevant, depicted in Figure 6-5. The broken live range a contains a hole consisting of {3,4,5,6}. Both c and d can be used to pad the hole. The original register conflict graph is shown top right; merging a with c and merging a with d yields the two register conflict graphs depicted at the bottom of the figure - they are identical. Note also that all three conflict graphs are 3-colorable, in other words, no merge operation causes an increase of the chromatic number of the derived graphs. The reason why the choice of c or d for merging with a is irrelevant is that the live ranges that overlap with c and d form a clique in the register conflict graph, consisting of nodes $\{c,d,b,e\}$. We call nodes like c and d *perfect matches* for the hole in a.

A live range that fits into a hole h may itself contain a hole h_{fit}. Therefore, live ranges that fit into h_{fit} fit into h as well. In that situation, the sequence in which h and h_{fit} are padded can determine whether or not h



Original conflict graph

Merging a and d

Merging a and c

Figure 6-4: Different conflict graphs for different merge operations

or h_{fit} can be eliminated via node merging. Our goal is to keep node merging to eliminate holes simple, so we avoid this problem by requiring continuity of a perfect match for a hole. To ensure that interactions of perfect matches with other live ranges remain "local" to the hole, we enforce that every live range that overlaps with a perfect match but not with the broken live range must also be a fit for the hole. To avoid backtracking when a fit for a hole is chosen for node merging, we require that the adjacency lists of two overlapping fits for a hole are identical. After this motivation, we can define perfect matches formally. In the following, let *adj(m)* denote the set of live ranges that overlap with a live range *m*.

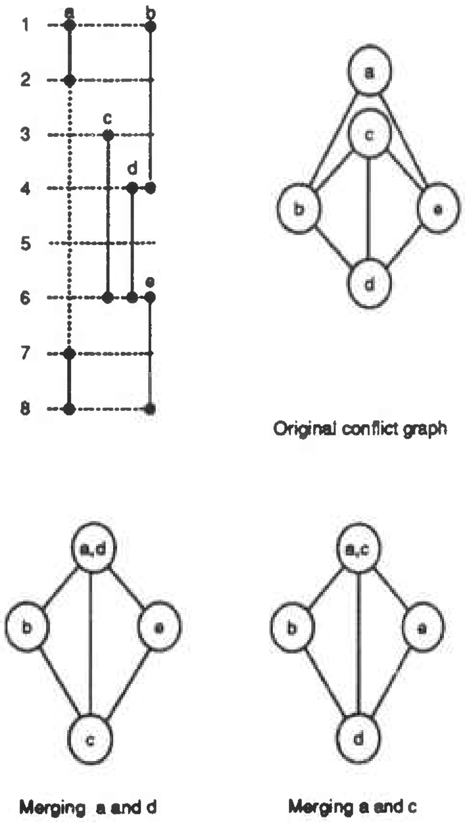
Definition 3: *(Perfect match for a hole)* Given a register conflict graph *G*, we say that a live range *m* is a perfect match for a hole *h* in a broken live range *l* iff

1. *m* is a fit for *h*
2. if $x \in adj(m) - \{adj(l) \cap adj(m)\}$ then *x* is a fit for *h*
3. if $x \in adj(m)$ and *x* is a fit for *h* then $\{adj(x) - m\} = \{adj(m) - x\}$

Examples of perfect matches are given in Figure 6-6. Live range *a* is broken and contains a hole $h = \{3,4,5,6,7\}$. Because *b,c* and *d* form a clique, and both *b* and *c* are fits for *h* both *b* and *c* are perfect matches for *h* - for the same reason, *e* is a perfect match for *h*.

Perfect matches have a number of desirable properties that allow to eliminate holes from broken live ranges without paying the penalty of needing additional colors to color the derived conflict graph. The first property of a perfect match *p* for a hole *h* of a broken live range *l* is that *p* and *l* can be merged while maintaining the chromatic number of the original conflict graph in the derived conflict graph.

Lemma 4: Given a register conflict graph *G* with broken live range *l* with hole *h* and a live



Original conflict graph

Merging a and d

Merging a and c

Figure 6-5: Equal register conflict graphs for different merge operations

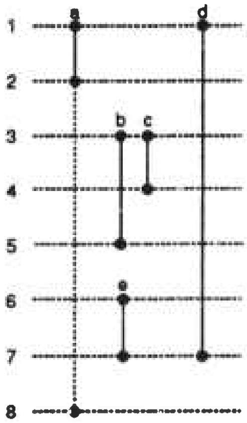


Figure 6-6: Perfect matches for a hole

range p such that p is a perfect match for h, let G' be the conflict graph derived from G by merging the nodes l and p. Then, the chromatic number of G' is equal to the chromatic number of G.

Proof: We show that the chromatic number of the two graphs are the same because a coloring for G' can be mapped to G and a coloring of G can be used to produce a valid coloring of G' without adding any colors.

Case 1: mapping a coloring of G' onto G: Let $p \cup l$ denote the node derived by merging live ranges p and l. A valid coloring for G' can be expanded into a valid coloring for G by using the color used for $p \cup l$ in G' for both p and l in G and by copying the colors for all other nodes in G' to the corresponding nodes in G.

Case 2: mapping a coloring for G onto G': Let c_x denote the color used for a live range x in a coloring of G. To map a coloring of G to G', we first change the coloring of G such that p is colored with c_l. If there is a live range $x \in \{adj(p) \cap adj(l)\}$, then $c_x \notin \{c_l, c_p\}$, so altering the color for p to c_l has no influence on x. Given a live range $x \in \{adj(p) - \{adj(p) \cap adj(l)\}\}$ such that $c_x = c_l$, coloring p with c_l is not possible without changing the color for x. By prerequisite $\{adj(x) - p\} = \{adj(p) - x\}$. Therefore, $y \neq p \in adj(x) \rightarrow c_y \neq c_p$. Hence, x can be colored with c_p. Therefore, the color for p can be changed to c_l and the coloring of the nodes in $adj(p)$ can be adjusted by exchanging c_l and c_p - in other words, the number of colors is not increased by coloring p and l identically. Once l and p are colored identically, merging nodes l and p can be performed without adjusting coloring - therefore, the chromatic numbers of G and G' must be equal.

Given a perfect match p for a hole h of a broken live range l and a live range x that is a fit for h such that $x \in adj(p)$, it is easy to see that x is also a perfect match for h and that it is irrelevant whether x or p is chosen for merging with l - in the example shown in Figure 6-5, the derived register conflict graphs for merging c with l and d with l are equivalent.

The next property of perfect matches is that the order in which perfect matches of the same hole h are merged is irrelevant - the remaining perfect matches are perfect matches of the holes of the newly derived merged live range. Before we formalize this in the next lemma, we go through an example. Figure 6-7 shows a broken live range a with perfect matches b, c and e. Because the adjacency set of b and c are equal, it is irrelevant which of the two is chosen to be merged with a, so we chose b. The picture to the right shows the same set of live ranges, except a and b have been merged into one live range now. Because c was adjacent to b before, c is now adjacent to the newly derived live range $a \cup b$ and can no longer be a perfect match. The original hole of live range a has now shrunk to the basic blocks *{6,7}*, and e is still a perfect match for the newly derived hole.

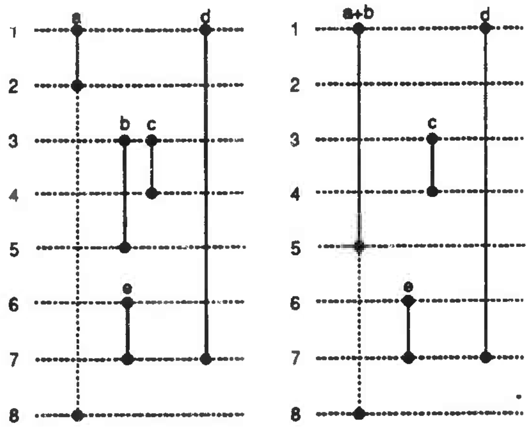


Figure 6-7: Sequences of perfect matches

Lemma 5: Given a flow graph with basic blocks B and register conflict graph G with broken live range l with hole h and live ranges p and p' that are both perfect matches for h, let $l \cup p$ be the live range derived from merging l and p. If $p' \cap (l \cup p) = \emptyset$, then p' is a perfect match for the remaining part of the hole $h' \subseteq h$ of the live range derived by merging l and p.

Proof: To show that p' is a perfect match for $l \cup p$, we first must show that p' is a fit for a hole in live range $l \cup p$. Because p' is a perfect match for h, $p' \cap B - h = \emptyset$. Because $p' \cap (l \cup p) = \emptyset$, $l \cup p$ must have a hole h' such that p' is a fit for h'. A live range $x \in (adj(p') - adj(l \cup p))$ must be a fit for h' as well, because by prerequisite x is continuous.

It remains to be shown that $\{adj(p')-x\}=\{adj(x)-p'\}\forall x\in adj(p')$ such that x fits h'. A live range $x\in\{adj(p')-adj(l)\}$ can not be adjacent to p, because all fits of h adjacent to p share the same adjacency list, but p' is not adjacent to p by prerequisite. Therefore, either $x\in adj(l)$ or $x\in adj(p')$. Thus merging p and l does not affect the fits of h' that are adjacent to p', and by prerequisite $\{adj(p')-x\}=\{adj(x)-p'\}$. Hence, p' is a perfect match for h'.

Lemma 5 states that the order in which which perfect matches are merged witn a broken live range does not matter at all - the graph derived by node merging with perfect matches is identical for all choices of perfect matches. In other words, no search is necessary to choose a live range for node merging; node merging is kept simple and inexpensive. Once all perfect matches are used up and the hole is filled, the derived conflict graph is still equivalent to the original conflict graph.

6.2.2. Imperfect matches and breaks

The restrictions that must be met by perfect matches were chosen such that the choice which perfect match was merged with a broken live range did not matter at all. In other words, merging of a broken live range with perfect matches for its holes can be done blindly. If we are willing to use more expensive algorithms to choose live ranges for padding holes, we can relax the restrictions that must be met by fits for a hole and still guarantee that the chromatic number of the derived conflict graph does not increase.

The motivation for our next definition is to chose the "largest" fit for a hole. Given a hole h and some fits for h, an imperfect match m for h is a fit whose adjacency list $adj(m)$ includes the adjacency lists of all fits in $adj(m)$. More formally:

Definition 6: *(Imperfect match for a hole)* Given a register conflict graph G, we say that a live range m is an imperfect match for a hole h in a broken live range l iff

1. m is a fit for h
2. If $x\in adj(m)-adj(l)$ then x is a fit for h
3. If $p\in\{x|x \text{ is a fit for } h\}$ then $adj(p)\cap adj(l)\subseteq adj(m)\cap adj(l)$

An example of an imperfect match is depicted in Figure 6-8. Live range a contains a hole and live ranges c, d and e are fits for that hole. The set of nodes adjacent to a consists of $\{b,f\}$. Because c is adjacent to both b and f, c is an imperfect match for a's hole. Neither d nor e are imperfect matches for a's hole.

Similar to Lemma 4, we can show that the graph derived by merging a broken live range with an imperfect match has the same chromatic number as the original graph.

Lemma 7: Given a flow graph with basic blocks B, register conflict graph G and a broken live range l with hole h, let p be an imperfect match for h, and let G' be the conflict graph derived from G by merging p and l. Then, the chromatic numbers of G and G' are identical.

Proof: It is easy to see that any coloring for G' can be expanded into a coloring for G without using additional colors. Given a coloring of G, let c_x denote the color for live range x in a coloring for G. Let $\{p_1,...,p_n\}$ be the live ranges that are fits for h. If $x\in adj(p_i)\cap adj(l), i\in\{1,...,n\}$, then $c_x\notin\{c_p,c_l\}$, hence x is unaffected by changing the color of p to c_l. Therefore, the only nodes affected by changing p's color to c_l are $\{p_1,...,p_n\}$. By prerequisite, no $p_i\in\{p_1,...,p_n\}$ is adjacent to l, therefore c_p and c_l can be exchanged among

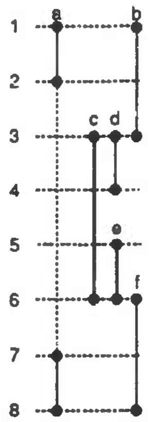


Figure 6-8: An imperfect match for a hole

$\{p_1,...,p_n\}$. Hence, a coloring of G can be used to produce a coloring of G' without introducing new colors - the chromatic numbers of both graphs are identical.

Imperfect matches do not have the nice properties of perfect matches. For example, given an imperfect match m for a hole h of a broken live range l, merging m and l might not fill up h completely. It is not guaranteed that the remaining hole can be padded with the remaining live ranges. Imperfect matches are useful however if they are "large" enough to fill the entire hole. In the example depicted in Figure 6-8, the imperfect match c fills the entire hole of live range a. The fact that c is an imperfect match allows us to deduce that by merging c and a the entire hole can be eliminated without increasing the number of colors needed to color the derived register conflict graph.

The reason why broken live ranges can create register conflict graphs that contain chordless cycles of size 4 or larger is that all fits for a hole h are interdependent - the color chosen for a live range f that is a fit for a hole h sometimes influences the choices of colors for all other fits for h. Since we are forced to use the same color for the *TOP* and the *BOT* part of a broken live range, chordless cycles are introduced in a register conflict graph. The motivation for the next definition is to determine basic blocks inside a hole h that partition the fits for h into independent sets. If there is such a basic block, the fact that we must color *TOP* and *BOT* parts of a broken live range identically has less impact.

So rather than restricting live ranges that are fits for a hole, we now define restrictions on the hole itself. We will show later that holes that meet these restrictions can be padded by their fits without compromising optimality of the overall coloring.

If there is a basic block b that is *not* contained in any fit for the hole, then b can be used to partition the fits for the hole into two sets: the fits that consist of basic blocks that consist of basic blocks that precede b and those that consist of basic blocks that succeed b. If in addition the only live ranges that contain b must also contain *all* basic blocks of the hole, no dependencies exist between the fits in those two sets. We will show that the absence of dependencies can be exploited to eliminate the hole. We first give a formal definition of a break in a hole.

Definition 8: *(Break in a hole)* Given a register conflict graph G with a broken live range l that contains a straight line hole $h=\{h_1,...,h_n\}$, we say that $h_i \in \{h_1,...,h_n\}$ is a break for h iff every live range that contains h_i contains $\{h_1,...,h_n\}$, i.e. all basic blocks that form the hole.

Figure 6-9 gives an example of a break in a hole. Live range *a* contains a hole consisting of basic blocks $\{3,4,5,6,7\}$. Basic block 6 is a break for that hole - the only live range that contains 6 is *f* which contains basic blocks of both the *TOP* and the *BOT* part of *a* - $\{1,2\}$ and $\{8\}$ respectively. The set of fits for the hole consists of $\{b,c,d,e\}$. The break partitions the live ranges that are fits for the hole into two independent sets - one "above" the break, $\{b,c,d\}$ and one "below" the break, $\{e\}$.

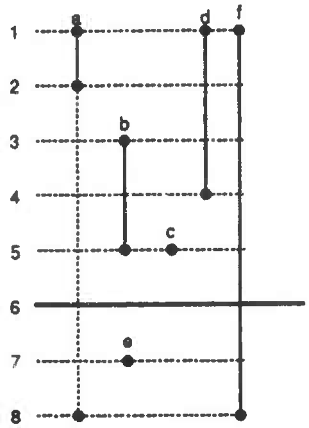


Figure 6-9: Break of a hole

A break in a hole is desirable; the intuitive reason is that the live ranges that are fits for that hole can be partitioned into two independent subsets via the basic block that is the break. The live ranges in one of the subsets can only be adjacent to the *TOP* or *BOT* part of the broken live range, but not both. Therefore, both parts can be colored independently and combined by renaming. This is stated in the next lemma.

> **Lemma 9:** Given a flow graph *F* with register conflict graph *G* such that there exists a backarcfree path that contains *all* basic blocks in *F*, let *l* be a broken live range that contains *one* hole *h* with break b_k. If all live ranges in *G* except *l* are continuous, *G* can be colored optimally in polynomial time.

The proof of Lemma 9 consists of three parts. In the first part we show that the flow graph *F* can be simplified to a flow graph *F'* that is equivalent for the purpose of register allocation. In the second part of the proof, we consider only the portion of *F'* that consists of the loop or conditional that causes the hole, called *F''*, and show how to color the conflict graph *G''* for *F''* optimally in polynomial time. The third part of the proof deals with extending the coloring for *G''* to a coloring for the original register conflict graph. As we go through the proof, we will illustrate our steps by examples. During the proof, we assume that the hole containing the break is caused by a loop. The case in which the hole is caused by a conditional can be shown analogously.

Proof: *Part 1: constructing F'*: W.l.o.g. assume that the hole *h* is caused by a loop *L* with loop head *a* and loop exit *e*. Let *F'* be the graph derived from *F* by removing the following edges:

1. all backarcs except that of *L*
2. all edges from the split node of a conditional *C* to the join node of *C*.

By Lemma 15 the register conflict graph of *F'* is equal to the original register conflict graph *G*.

Part 2: constructing G': Let $b_1,...,b_n$ be the set of basic blocks that form the hole *h*, and let *F''* be the subgraph induced by $\{b_0,b_1,...,b_n,b_{n+1}\}$, where b_0 is b_1's predecessor in *F'*, and b_{n+1} is b_n's successor in *F'*. Because *l* is the only broken live range, $b_k \in \{b_1,...,b_n\}$, and both b_0 and $b_{n+1} \in l$. *G'* consists of those live ranges in *F'* that do not contain the break b_k, in other words live ranges that contain a subset of the following basic blocks: $\{b_0,...,b_{m+1}\} - \{b_k\}$.

The construction of G' is depicted in Figure 6-10. On the left side of that figure the original conflict graph G is shown. The broken live range is a, and the "straight line" hole of a consists of basic blocks {5,6,7,8,9}. Basic block 8 forms a break in the conflict graph. G' is shown on the right side of Figure 6-10 - a now consist only of basic blocks {4,10}, the two blocks that "define" the hole. Only live ranges that either fit into the hole or that contain either 4 or 10 are included in G'.

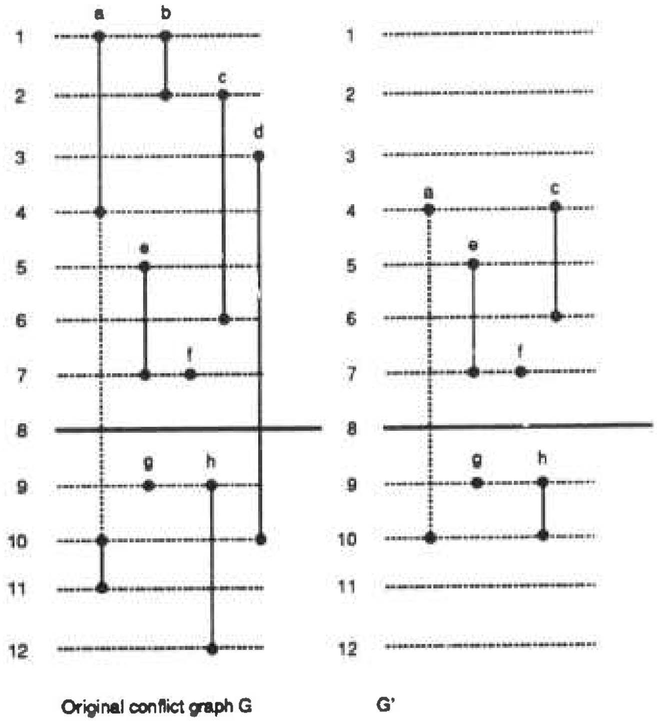


Figure 6-10: G and G'

Because b_k is a break in h, no live range in G' contains b_k. Then, G' can be partitioned into two subgraphs, G_1 induced by those live ranges that consist of basic blocks among the "first half" of h, $\{b_0,...,b_{k-1}\}$ and G_2 induced by the live ranges that consist of basic blocks in the second half of h, $\{b_{k+1},...,b_{n+1}\}$. The construction of G_1 and G_2 is depicted in Figure 6-11. Note that both G_1 and G_2 contain a part of the broken live range l, b_0 and b_{m+1} respectively. Because $\{b_0,...,b_{k-1}\}$ and $\{b_{k+1},...,b_{n+1}\}$ both form straight line code and no live range in G_1 and G_2 contains holes, G_1 and G_2 must be interval graphs. Hence, both G_1 and G_2 can be colored optimally in polynomial

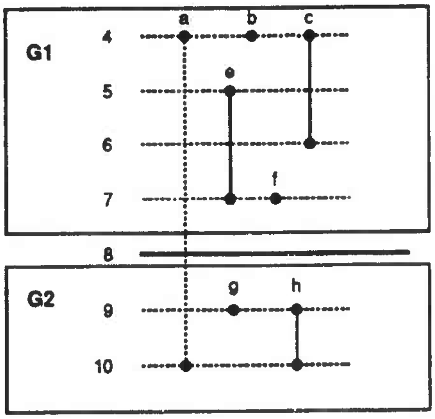


Figure 6-11: G_1 and G_2

time. Let c_{TOP} be the color used to color the part of l consisting of b_0 in G_1 and c_{BOT} the color used to color the part of l consisting of b_{m+1} in G_2. An optimal coloring for G' can be derived from the colorings for G_1 and G_2 by exchanging c_{TOP} and c_{BOT} in G_2. Implicitly, h can be eliminated by padding h with all live ranges in G' that are colored with c_{TOP} and merging the appropriate live ranges with l in G'. After those merge operations, l is equivalent to a continuous live range.

Part 3: deriving a coloring for the original register conflict graph: Deriving a coloring for the original register conflict graph G is now straightforward. First, we add to G' every live range v that contains b_k. Because b_k is a break, v must overlap with *every* live range in G', hence a new color must be used for v.

This is depicted in Figure 6-12. In our example, we must add the live range d. Because all

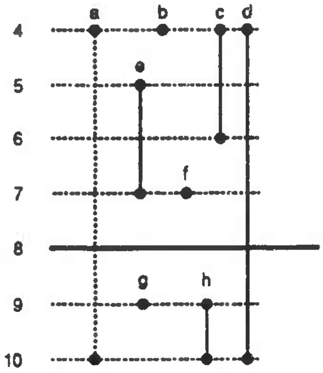


Figure 6-12: Adding live ranges that contain the break

remaining live ranges of G are continuous and the original program is equivalent to straight line code, the live ranges that contain b_0 must form a clique that separates G' from the "top part" of G. Likewise, the live ranges that contain b_{n+1} must form a clique that separates G' from the "bottom third" of G. The two clique separators are shown in Figure 6-13. Hence, the optimality of the coloring for G' is maintained when an overall coloring is derived from individual optimal colorings of the top part of G, G' and the bottom part of G - q.e.d.

Note that if there exists a break in a straight line hole, no restrictions must be met by live ranges that are fits for the hole. The prerequisites of Lemma 9 may seem strict; note though that when register conflict graphs are simplified hierarchically, programming constructs that are nested inside the basic blocks that form a hole will have been processed by the time the hole is treated. Therefore our techniques to collapse or sequentialize branch clauses and to eliminate holes nested inside have already been applied, with the effect that the enclosing hole can be viewed as if it were a straight line hole for the purpose of register allocation.

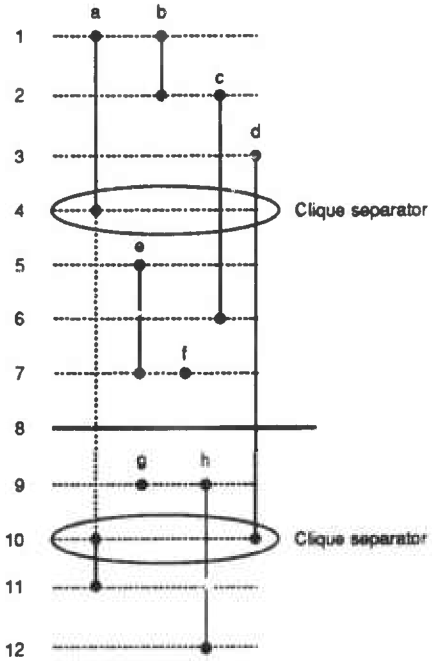


Figure 6-13: Clique separators to remaining conflict graph

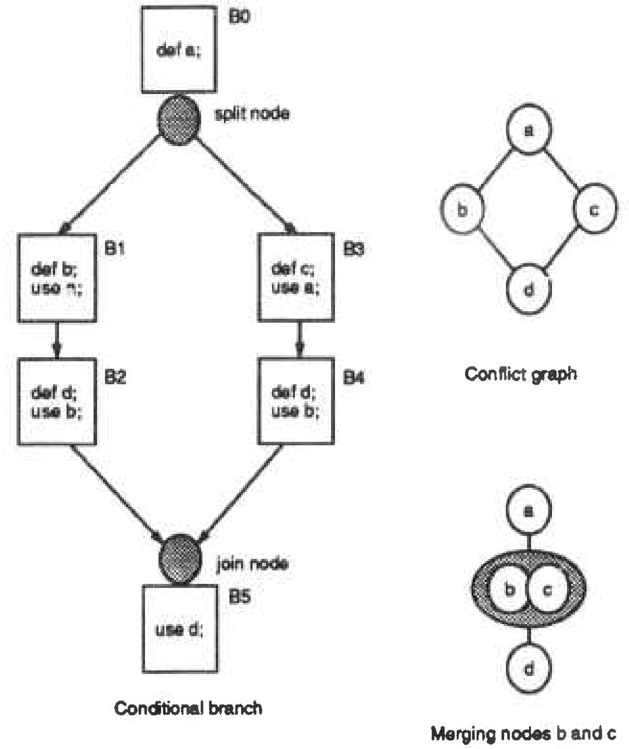
6.3. Node merging and conditionals

In the previous chapter we showed that register conflict graphs in which all backarc and *BGLOBAL* live ranges are continuous, and in which all *LOEN* and *LOEX* live ranges overlap in each branch sibling, are interval graphs and can be colored optimally in polynomial time.

For the remainder of this chapter we assume that a conditional branch contains no conditional broken live ranges. Even in the absence of broken live ranges, there are cases of conditionals that have register conflict graphs that are *k* colorable, but not interval graphs. In the following sections we examine types of register conflict graphs for conditionals that are not interval graphs and therefore in general can not be colored optimally by the node removal technique. We discuss how node merging can be used to simplify register conflict graphs for conditional branches, and how in some cases node merging can change a non-interval register conflict graph for a conditional into an interval graph.

6.3.1. Merging of E_1 related nodes

Figure 6-14 depicts an example of a conditional with a register conflict graph for which the node removal technique is unable to produce a 2-coloring. The conditional branch consists of the split and join node, and two branch clauses, *{B1;B2}* and *{B3;B4}* respectively. Variable *a* is live at the split node and dead at the join node, hence *a* is a *LOEN* live range in that branch. Variable *d* is live at the join node, and dead at the split node, hence *d* is defined inside the branch clauses and a *LOEX* live range. Variables *b* and *c* are local inside the branch clauses and are therefore *BLOCAL* live ranges.



Conditional branch

Conflict graph

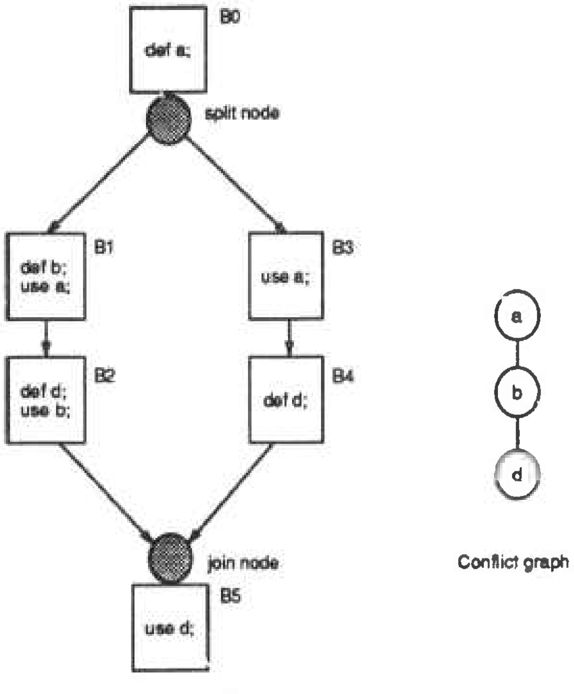
Merging nodes b and c

Figure 6-14: Conditional branch with *LOEN* and *LOEX* live ranges and conflict graph

The register conflict graph for the conditional branch is depicted to the right of the flow graph. The register conflict graph for the conditional branch is 2-colorable and not recognized as such by the node removal technique. Since *b* and *c* are live in distinct branch clauses, the color chosen for node *b* is independent of the color chosen for node *c* in the register conflict graph. A 2-coloring of the graph depends on the colors chosen for *b* and *c* - they must be identical. The knowledge that *b* and *c* live only in distinct branch clauses can be used to determine that it is safe to color *b* and *c* with the same color. Like in register conflict graphs for loops, nodes in register conflict graph for conditional branches can be merged as long as it is safe to do so. This is depicted in the bottom right of Figure 6-14. The reason why *b* and *c* could be safely merged into one single node is that they are both adjacent to exactly the same set of nodes in the register conflict graph, *a* and *d*.

> **Definition 10:** *(The E_1 relation)* Given the register conflict graph *G* of a conditional branch *C*, let c_1 and c_2 be two live ranges that are *BLOCAL* w.r.t. *C*. We say that c_1 and c_2 are *E_1 related* iff c_1 and c_2 occur in different branch clauses and $adj(c_1)=adj(c_2)$.

It is easy to see that two *BLOCAL* nodes that are E_1 related can be colored identically and thus merged into one node. The idea behind merging E_1 related nodes is to re-use the same color in different branch clauses for *BLOCAL* live ranges that are by definition local to one branch clause. The graph *G'* derived from the original register conflict graph *G* by merging two E_1 related nodes *x* and *y* is equivalent to the register conflict graph of a conditional branch in which all instructions that use or define the variable with live range *x* (or *y*) are eliminated.



Conditional branch

Conflict graph

Figure 6-15: Register conflict graph of branch derived from Figure 6-14 by eliminating definitions and uses of variable *c*

An example is depicted in Figure 6-15. The live ranges for *b* and *c* are E_1 related, and merged in the original register conflict graph (Figure 6-14). The branch depicted in Figure 6-15 is identical to that in Figure 6-14, except that the definition of *c* in block B3 and the use of *c* in block B4 have been removed. The register conflict graph is shown to the right and is identical to the register conflict graph containing the merge node depicted in Figure 6-14.

The observation that one of two E_1 related live ranges can be eliminated both from the register conflict graph and from the program without compromising a *k*-coloring of the register conflict graph is formalized in the next lemma.

> **Lemma 11:** Given a register conflict graph *G* of a conditional branch *B* that contains two E_1 related *BLOCAL* live ranges *x* and *y*, let *G'* be the register conflict graph derived form *G* by merging the nodes representing *x* and *y*. Then, the chromatic numbers of *G'* and *G* are equal.
>
> **Proof:** W.l.o.g. assume that the candidate for removal is *x*. The graph derived from *G* by eliminating the node *x* and all edges incident on *x* is equal to *G'*. Therefore a coloring for *G* can be directly mapped onto *G'*, omitting the color used for *x*. Because $adj(x) = adj(y)$, the color used for *y* can be used for the merge node (x,y). It is easy to see how a coloring for *G'* can be mapped onto *G*: the color used for (x,y) is used for both *x* and *y* in *G*. Therefore, the chromatic numbers of *G* and *G'* must be equal.

Note that eliminating node *x* from the register conflict graph *G* in Lemma 11 is equivalent to removing all instructions that define or use the variable *v* with live range *x* from the conditional. This is possible because *x* is a *BLOCAL* live range, and does not extend beyond the split or the join node of the conditional.

Since two E_1 related nodes are adjacent to the exactly same set of edges in the register conflict graph, the number of edges incident on the merge node is equal to the number of edges incident to the individual nodes that were merged. For each other node in the original graph, the number of incident edges is either the same or decreases. In other words, node merging of E_1 related nodes always simplifies the register conflict graphs.

In the example depicted in Figure 6-14, merging of E_1 related nodes turns the original register conflict graph into an interval graph. The reason is that *every BLOCAL* node is E_1 related to a *BLOCAL* node in *every* other branch clause. As a consequence, all *BLOCAL* live ranges can be merged with their E_1 partners in one ***unique*** branch clause. In the example depicted in Figure 6-15, the instructions that form live range *c* in the original branch depicted in Figure 6-14 are removed. Now, there exists only *one* branch clause that contains *BLOCAL* live ranges - all others contain only *LOEN*, *LOEX* or *BGLOBAL* live ranges.

The fact that all *BLOCAL* nodes can be eliminated by node merging does not necessarily mean that the graph derived by all merge operations is an interval graph. This is depicted in Figure 6-16. The live ranges of a conditional branch that consists of 3 straight line branch clauses are shown. For readability, we show only the live ranges rather than the entire flow graph.

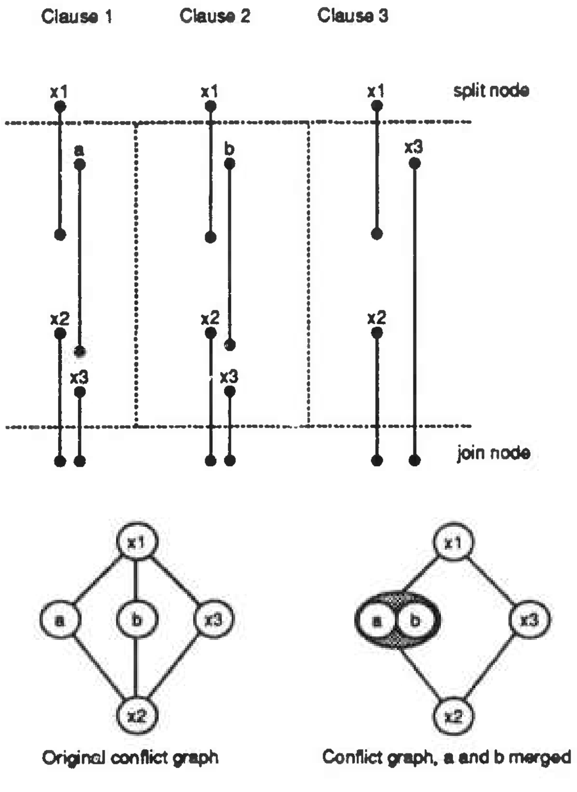


Figure 6-16: Non-interval register conflict graph after merging *all BLOCAL* live ranges

The branch contains one *LOEN* live range x_1, two *LOEX* live ranges x_2 and x_3 and two *BLOCAL* live ranges, *a* and *b* respectively. (Note that x_1, x_2 and x_3 occur in each branch clause, all occurrences form one single live range). Live ranges *a* and *b* are E_1 related - both are adjacent to x_1 and x_2, and can therefore be merged in the register conflict graph. The original register conflict graph is depicted to the bottom left, the conflict graph resulting from merging a and b is shown bottom right. Even though *all BLOCAL* live ranges are merged into a single live range, the resulting conflict graph is not an interval graph, because x_1 overlaps with x_3 in clause 3, but not in clause 1 or 2, where the *BLOCAL* live ranges occur.

To ensure an interval conflict graph, in addition to requiring that all *BLOCAL* live ranges can be merged into one unique branch clause it is needed that the subgraph formed by the *LOEN* and *LOEX* live ranges in each individual branch clause is identical. Figure 6-17 depicts *LOEN* live range x_1 and *LOEX* live ranges x_2 and x_3 in different branch clauses. Note that this example is identical to the example shown in Figure 6-16 with all *BLOCAL* live ranges removed. The register conflict graphs for the individual branch clauses are shown below the corresponding branch clause. Note that the graph in clause 3 differs from the graphs in clauses 1 and 2.

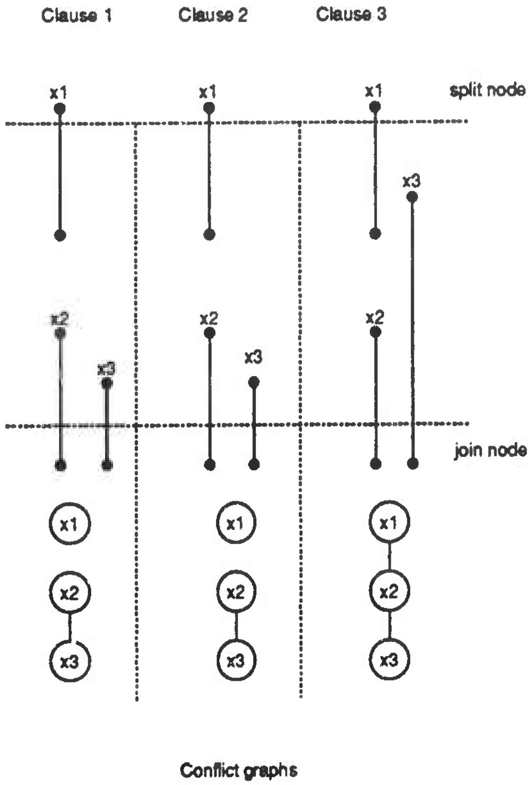


Figure 6-17: Register conflict graphs formed by *LOEN* and *LOEX* live ranges in individual branch clauses

If the register conflict graphs formed by *LOEN* and *LOEX* live ranges are identical in each branch clause, merging E_1 related live ranges a and b is equal to the register conflict graph of clause 1, shown in Figure 6-18. Since clause 1 consists of straight line code, the register conflict graph obtained by node merging is an interval graph. We formalize this in the next lemma.

Lemma 12: Let G be the register conflict graph of a conditional B with branch clauses $c_1,c_2,\ldots,c_n$, such that B contains no broken *BGLOBAL* live ranges, the register conflict graphs formed by the *LOEN* and *LOEX* live ranges are identical in each branch clause, and for each *BLOCAL* live range $b \in c_i \ \exists \ b' \in c_j \forall i,j \in \{1,\ldots,n\}$ such that b and b' are E_1 related. Then the graph G' derived from G by merging all *BLOCAL* live ranges with a node representing a live range in one unique branch clause is equivalent to the register conflict graph of that branch clause in isolation.

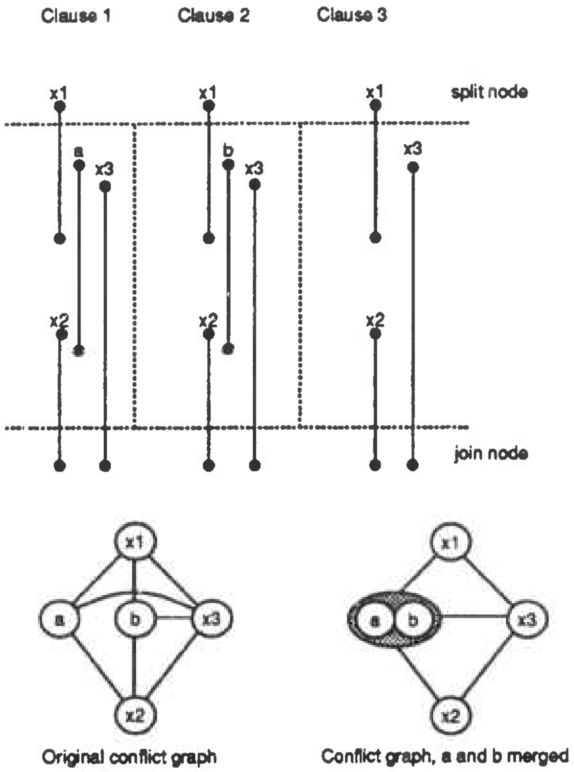


Figure 6-18: Interval register conflict graph after merging *all BLOCAL* live ranges

Proof: By Lemma 11, *G'* is equal to the register conflict graph of the conditional *B* by removing all instructions that form live ranges represented by nodes that are merged. Further, the register conflict graphs for the *LOEN* and *LOEX* live ranges are identical in each individual branch clause - hence the register conflict graph for the branch clause to which all *BLOCAL* live ranges have been merged is equal to *G'* - q.e.d.

In the next section we show how the strict E_1 relation can be relaxed in a way that still allows to use node merging to systematically simplify register conflict graphs of conditional branches.

6.3.2. Merging of E_2 related nodes

The E_1 relation is very restrictive. Nodes that are E_1 related must be *BLOCAL* w.r.t. a branch *B*. By definition two E_1 related nodes must be adjacent to the exactly same set of nodes - hence, those nodes can not be *BLOCAL* in the same conditional *C*. We can relax the E_1 relation by allowing adjacency to other (restricted) *BLOCAL* nodes, but still require that the set of adjacent nodes that are *LOEN*, *LOEX* or *BGLOBAL* be identical. More formally:

Definition 13: Given a conditional *B*, let *x* and *y* be two live ranges that are *BLOCAL* w.r.t. *B*, such that *x* and *y* occur in different branch clauses. We say that *x* and *y* are E_2 related iff

- All *BLOCAL* live ranges adjacent to *x* have the same adjacency lists:

  $\forall x' \in \{adj(x) \cap BLOCAL(B)\} : adj(x') - x = adj(x) - x'$

- All *BLOCAL* live ranges adjacent to *y* have the same adjacency lists:

$$\forall y' \in \{adj(y) \cap BLOCAL(B)\} : adj(y') - y = adj(y) - y'$$

- The set of non-*BLOCAL* live ranges adjacent to x is equal to the set of non-*BLOCAL* live ranges adjacent to y:

$$adj(x) \cap G = adj(y) \cap G, \text{ where } G = LOEN(B) \cup LOEX(B) \cup BGLOBAL(B)$$

Note that E_1 related nodes are also E_2 related. Figure 6-19 shows the live ranges in the branch clauses of a conditional branch, consisting of *LOEN* live range x_1, *LOEX* live range x_2 and *BLOCAL* live ranges a, a', a'', b and b'. Live ranges a, a' and a'' are E_2 related to live ranges b and b'.

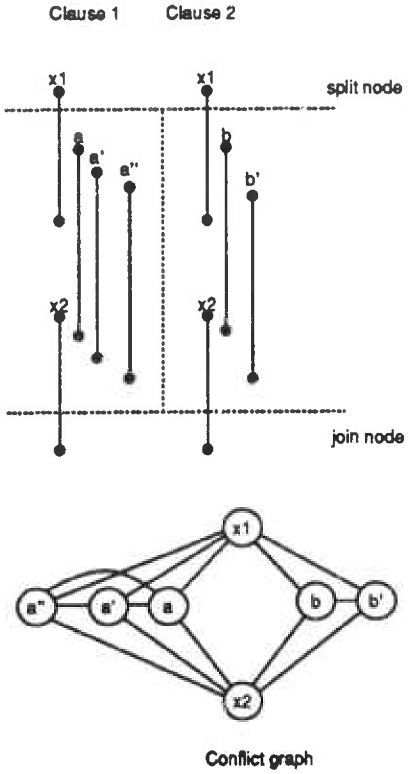
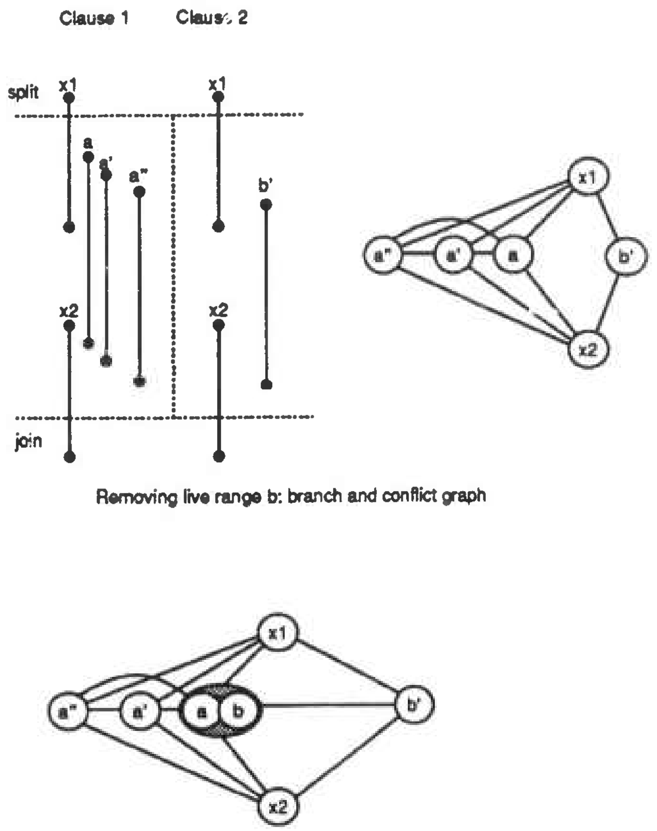


Figure 6-19: E_2 **related nodes**

If in a conditional branch all *BLOCAL* live ranges of one branch clause are E_2 related to a *BLOCAL* live range in every other branch clause, we can show by arguments similar to those used in Lemma 12 that merging all E_2 related nodes results in a graph that is equivalent to the register conflict graph of one single branch clause - given that the register conflict graphs formed by *LOEN* and *LOEX* live ranges are identical in each individual branch clause.

Things are a bit more complicated, because Lemma 11 does not hold for two E_2 related nodes. This is demonstrated in Figure 6-20. Nodes a and b of the original conditional shown in Figure 6-19 are merged into one single node, and the resulting graph is depicted to the bottom. Unlike for E_1 related nodes, the graph obtained by merging a and b is not equivalent to the register conflict graph of the conditional in which live range b has been removed. The conditional branch in which live range b has been removed is shown top left (Figure 6-20). The register conflict graph for that conditional branch is depicted top right, and differs from the graph at the bottom.



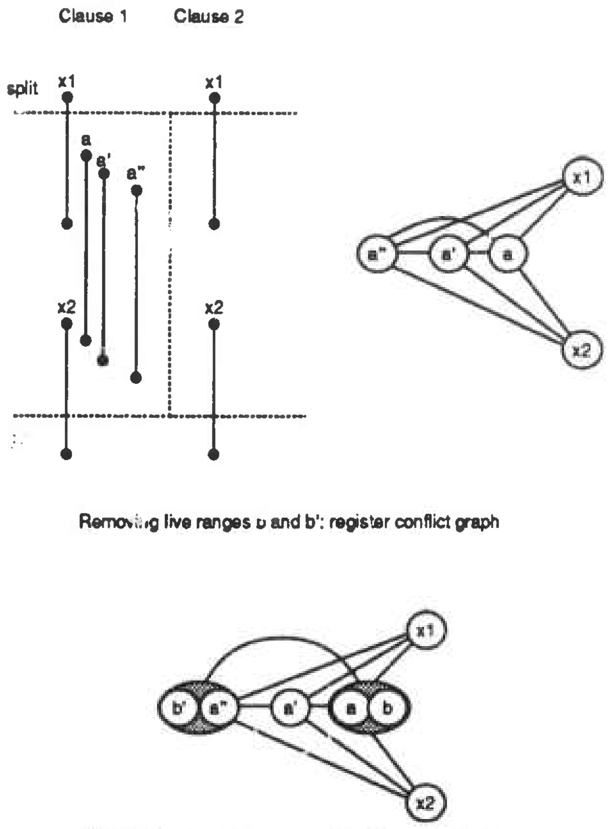
Removing live range b: branch and conflict graph

Merging a and b in original register conflict graph

Figure 6-20: Difference between removing live range b and merging it with a in original conflict graph

The reason why Lemma 11 does not work for E_2 related nodes is that E_2 related nodes can be adjacent to other *BLOCAL* live ranges within the same branch clause. Merging two E_2 related nodes n_1 and n_2 causes (by symmetry) n_2 to move to a different branch clause. Other *BLOCAL* live ranges adjacent to n_2 are now adjacent to the merged node consisting of (n_1,n_2), and therefore an edge is introduced between *BLOCAL* nodes of different branch clauses. In the example depicted in Figure 6-20, the node that contains a and b is adjacent to the node b' - the merge node "belongs" to clause 1 and b' belongs to clause 2. If however *all BLOCAL* live ranges in the clique formed by E_2 related nodes are merged, the resulting graph is equivalent to the conflict graph of a branch in which *all* members of the clique have been removed, shown in Figure 6-21. The conditional branch in which both b and b' have been removed is shown to the top left, and the corresponding register conflict graph is shown to the right. This conflict graph is identical to the graph obtained from the original register conflict graph by merging all *BLOCAL* members of the clique that contains b in clause 2 with their E_2 partners of clause 1.

In the example we used in Figure 6-21, nodes b and b' were merged with their E_2 partners in clause 1. Since the number of *BLOCAL* live ranges adjacent to a is larger than the number of *BLOCAL* live ranges adjacent to b, there were enough nodes in clause 1 such that each *BLOCAL* node adjacent to b could be merged. This is not the case if we wished to merge the other way around: merging three live ranges a,a' and a'' with two live ranges b and b' does not work - and the trick by mapping the resulting graph to the register conflict graph of one branch clause does not work either. This is depicted in Figure 6-22.



Removing live ranges a and b': register conflict graph

Merging a,b and a",b' in original register conflict graph

Figure 6-21: Merging all clique members of one branch clause

The conditional branch in which live ranges *a* and *a'* have been removed is shown to the top left, and the corresponding register conflict graph to the top right. Note that the register conflict graph differs from the graph derived by merging *a* and *b* and *a'* and *b'* - the adjacency of *a''* with the merge nodes is lost if live ranges *a* and *a'* are simply removed. We can avoid this problem by adding a dummy live range *d* that consists of the same basic blocks as *b*. Note that adding the dummy live range *d* to the register conflict graph is equivalent to duplicating every definition and use of the variable with live range *b* by the definition or use of a dummy variable. Hence, we have constructed a new conditional *B'* such that the register conflict graph for *B'* is identical to the graph obtained from the original register conflict graph by merging *a* and *a'* with *b* and *b'* respectively. This is shown in Figure 6-23.

> **Lemma 14:** Given a conditional *C* with conflict graph *G* that consists of branch clauses $c_1,...,c_n$ such that the graph induced by the conflict graph of an individual branch clause c_i and the *LOEN* and *LOEX* live ranges is identical for each $c_i \in \{c_1,...,c_n\}$, let *v* and *v'* be two *BLOCAL* live ranges of different branch clauses c_i and c_j such that *v* and *v'* are E_2 related. The chromatic numbers of *G'* derived from *G* by merging *v* and *v'* are identical.

> **Proof:** Clearly, a valid coloring for *G'* can be easily extended to a valid coloring for *G* by directly mapping the colors of all nodes to the same nodes in *G* and by mapping the color of the merged node (*v*,*v'*) to both *v* and *v'*. Given a coloring for *G*, we obtain a coloring for *G'* by enforcing the same color for *v* and *v'*. Let col_v and $col_{v'}$ be the colors used for *v* and *v'* respectively. Because every *LOEN*, *LOEX* or *BGLOBAL* live range that is adjacent to *v* is also adjacent to *v'*, no global live range is colored with col_v or $col_{v'}$. Let *x* be a *BLOCAL* live range adjacent to *v*. By definition of the E_2 relation, all *BLOCAL* live ranges adjacent to *v* have

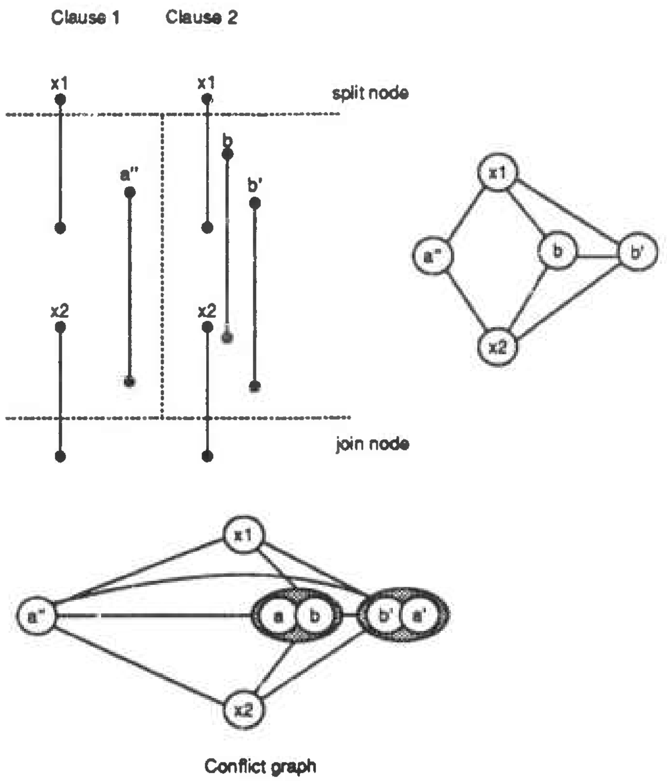


Figure 6-22: Merging larger clique of one branch clause with smaller clique of other branch clause

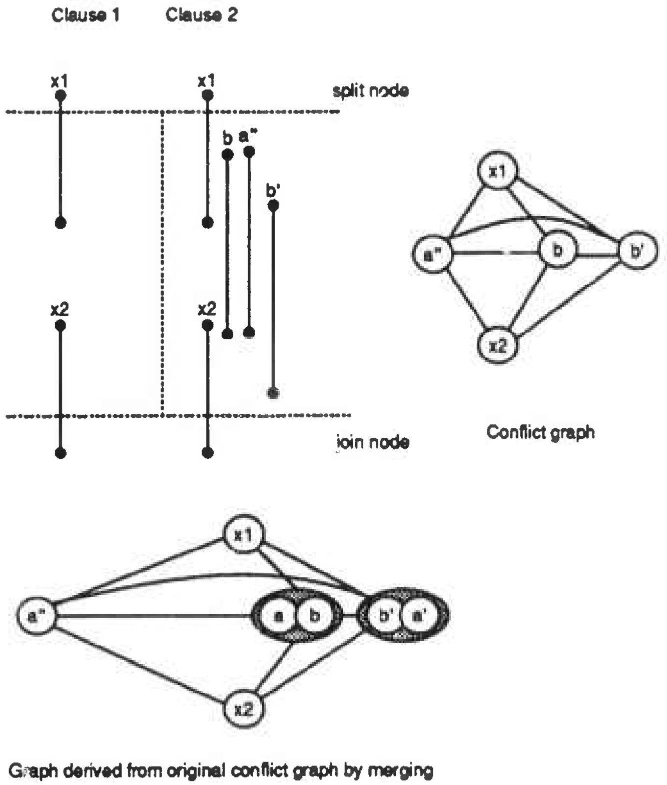


Figure 6-23: Merging larger clique of one branch clause with smaller clique of other branch clause

identical adjacency sets. Hence, the colors col_v and $col_{v'}$ can be exchanged in the live ranges that are adjacent to v - no new colors are needed.

Note that if all *BLOCAL* live ranges can be merged with *BLOCAL* live ranges of one unique branch clause, the derived register conflict graph is equivalent to the register conflict graph of just one branch clause. Consequently, E_1 and E_2 related merging can be used to collapse several branch clauses into one - in the case of non-nested conditionals, this leads to straight line code and hence to an interval register conflict graph.

6.4. Chapter summary

We have seen that node merging can be applied to broken live ranges and to *BLOCAL* live ranges of conditionals. In some cases it is even possible to change a non-interval register conflict graph into an interval graph via node merging. Because the node removal technique is in general unable to produce a k coloring for k-colorable non-interval graph, there are cases of register conflict graphs for which the node removal introduces spill code that can be avoided altogether by node merging. Even in cases where it is not possible to eliminate all non-chordal cycles in the register conflict graph, node merging always results in a simpler register conflict graph.

The advantages of encoding structural knowledge in register conflict graphs are twofold. First, the performance of the node removal technique can be provably enhanced by node merging. Second, structural analysis permits to partition the register conflict graph into disjoint subgraphs that can be colored individually and the colorings of the individual parts can be joined to an overall coloring by renaming. Both aspects of structural knowledge in register conflict graphs are the focus of an experimental evaluation of our model for register conflict graphs, and is discussed in the next chapter.

Chapter 7

Structured Global Register Allocation in Practice: Development of a Parallel Framework

Developing a parallel framework for global register allocation required the analysis of register conflict graph. We have shown that this is possible within our structured model described in the previous chapters. Initially motivated by parallelization, we find that our model and our analytical methods lead to a more thorough understanding of sequential global register allocation; once the location of clique separators is known, the parallelization of global register allocation comes naturally, almost as a "byproduct" of our analysis. The success of the parallelization depends on the knowledge collected by our method. In this chapter, we therefore concentrate on the evaluation of our *method*.

Our method is best described as analysis of register conflict graphs prior to the assignment of specific registers to live ranges in the conflict graph. The purpose of this analysis is to map a register conflict graph to an equivalent interval graph and, if this is not possible, to identify which portions of a register conflict graph are "responsible" for an overall non-interval register conflict graph. Mapping a register conflict graph to an equivalent interval graph is done via mapping a well structured flow graph to a straight line flow graph that is equivalent for the purpose of global register allocation. By Theorem 11 of Chapter 5, the straight line sequence of basic blocks can be used to enumerate all cliques in the register conflict graph in the .<. order described in Chapter 5.

Interval register conflict graphs are desirable for three main reasons. First, in an interval graph *all* clique separators can be found in polynomial time. Second, interval graphs can be colored optimally in polynomial time. Third, by enumerating all cliques of an interval graph in the .<. order it is possible to identify regions of the conflict graph that require spilling: given *k* registers, live ranges that occur in cliques that are larger than *k* must be considered for spilling to memory or for splitting. The important fact is that this is known *before* the actual coloring.

The success of our method depends on how often our transformations of flow graphs and conflict graphs are applicable to real programs and on the number of separator cliques that can be located. If large portions of a register conflict graph can be mapped to equivalent interval graphs, we can locate a large number of separator cliques. Knowledge about separator cliques is the basis for parallelizing global register allocation but is particularly well suited to improve sequential register allocation.

Given the knowledge about the location of clique separators, the register conflict graph can be partitioned into clique connected components that are either interval graphs or non-interval graphs. In that partition, the non-interval graphs are formed by live ranges that occur in conditionals that could not be collapsed or linearized, or by live ranges that contain basic blocks that occur in a hole of a broken live range that could not be eliminated. In other words, our structured method enables us to concentrate on small portions of the register conflict graph that are "responsible" for an overall non-interval register conflict graph. Because such hot spots in a register conflict graph typically consist only of a small subset of the live ranges, expensive heuristics or even exhaustive search can be used to find an overall good or optimal coloring. Isolating regions of the register conflict graph in which spilling is necessary is useful because more knowledge can be used in spilling heuristics. Further, the register conflict graph can be partitioned hierarchically via clique separators. Clique separated critical portions of the register conflict graph can be colored independently and combined with the remaining conflict graph without compromising optimality of the individual coloring.

To evaluate the performance of our method for analyzing and partitioning register conflict graphs for well structured programs, we implemented our techniques and measured its performance on a set of benchmark kernels. In this chapter, we present our implementation and report the results of our evaluation.

7.1. Evaluation of our method

Our method pre-processes the register conflict graph so that the subsequent register coloring can use the knowledge gathered during the analysis and operates on a simplified register conflict graph. To evaluate our method for practical purposes, we have to assess how applicable our flow graph and register conflict graph transformations are for real flow graphs. The transformations of our method achieve three goals:

- The simplification of a structured program into a piece of straight line code that is equivalent for the purpose of register allocation
- The simplification of a non-interval register conflict graph to an equivalent interval graph
- The detection of separator cliques for the parallelization of global register allocation

Transformations of flow graphs and conflict graphs:

In the previous chapters we have developed a set of restrictions that must be met by the live ranges of a program if our simplifications can be carried out without compromising optimality of global register allocation. The first goal of an experimental evaluation of our method is to determine how often this set of restrictions is met by real programs. For programs that can not be reduced to straight line code for the purpose of register allocation, we determine the size of the portions that could not be reduced to straight line code.

Parallelization:

Given the results of our graph transformations, we can identify separator cliques in the register conflict graph. We demonstrate the potential for parallelization by measuring the size of the ordered sequences of separator cliques in the register conflict graph of each benchmark kernel. We discuss the tradeoffs between choosing small separator cliques and creating parallel tasks of even sizes.

7.1.1. Implementation

Registers are a scarce resource on every computer architecture [Pat/Hen 90]. The need of registers is further increased by global optimizations that introduce temporaries that have to reside in registers. Conflict graphs for globally optimized programs are in general much more complicated than conflict graphs produced by compilers that do not employ global optimizations. For this reason we chose as input to our program intermediate code produced by an optimizing compiler. The C compiler for the IWARP machine fits this category and was used to produce the input to our conflict graph analyzer. The goal of our implementation was to evaluate two aspects of the method. First, to determine if the restrictions that permit an interval register conflict graph are met by realistic user programs, and second to assess the effectiveness of a parallelization of global register allocation based on clique separators.

The gist of our method is the *analysis* of register conflict graph, and our techniques aim for the *simplification* of register conflict graphs. To assess how often such simplifications could be carried out on the flow- and conflict graphs of realistic programs, it was sufficient to run our analysis "off line", and we did not incorporate our register allocator into the backend of the IWARP compiler.

Our implementation consists of 3 phases. Phase 1 is the "setup" phase, and consists of extracting the parse tree, the register conflict graph and the live ranges of variables from the input program in intermediate form. The *analysis* and *simplification* of the register conflict graph are performed during Phase 2. Phase 3 conducts the data partitioning for the parallelization of global register allocation. This includes determining the clique separators and the computation of the accumulated live ranges per basic block. Figure 7-1 gives an overview of the parts of our implementation. Because they form the gist of our implementation, we describe Phase 2 and Phase 3 in more detail in the next paragraphs.

7.1.2. Simplifying the conflict graph

The input to Phase 2 of the algorithm is the parse tree of a program, along with the live ranges and the register conflict graph. Note that each loop or conditional in the flow graph is represented by a node in the parse tree. Innermost constructs are the leaves of the tree, while the successors of the root of the parse tree represent the outermost programming constructs. The purpose of Phase 2 is to determine which basic blocks belong to conditionals or loops that could not be simplified to straight line code, called *prohibited blocks*. The nodes in the parse tree are processed in post order. The restrictions that are tested in our implementation were chosen such that every node in the parse tree must be processed exactly once. Hence our algorithm does not involve backtracking and runs in polynomial time. The detailed algorithm for Phase 2 is given in Figure 7-2.

At the end of Phase 2, every basic block that is not among *prohibited blocks* is part of straight line code, sometimes derived by removing backarcs from loops and by the sequentialization or the collapsing of conditionals. Hence, all live ranges that contain a block not in *prohibited blocks* form a clique that separates the conflict graph into two disjoint pieces. Phase 3 consists of computing the separator cliques for all but the prohibited basic blocks and the partitioning of the flow graph into clique separated components. The actions of Phase 3 are outlined in the algorithm given in Figure 7-3.

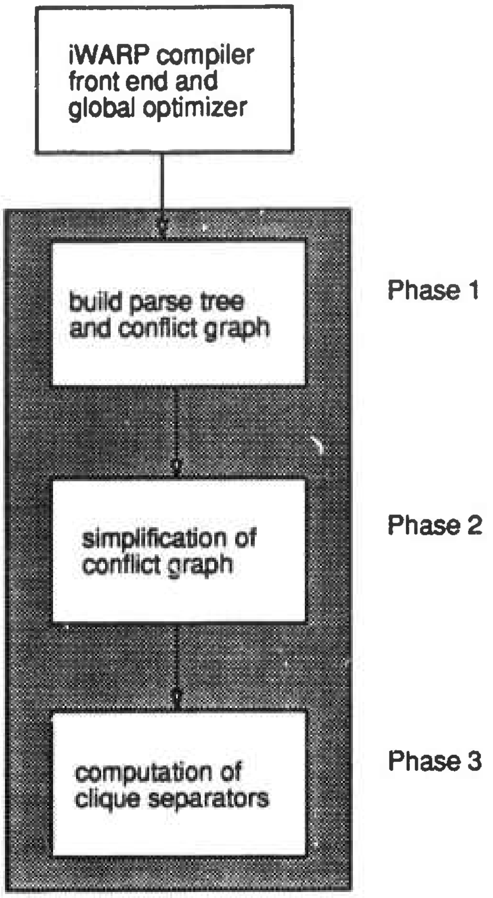

iWARP compiler front end and global optimizer
build parse tree and conflict graph
Phase 1
simplification of conflict graph
Phase 2
computation of clique separators
Phase 3

Figure 7-1: Phases of the structured global register allocator

In the evaluation of our method, we first concentrate on the effects of Phase 2, and analyze the parallelization in Phase 3 separately.

7.2. Input data

We measured the performance of our method for 33 benchmark kernels. The benchmark for our evaluation consists of the following functions:

1. The Livermore kernels [McMahon 86]
2. The larger examples of the Numerical Recipes collection [Press et al 88]
3. A collection of programs from the WARP library [Pomerleau, et al. 88, Kung 88]

Because our method requires well structured programs, we had to modify the original code slightly in some cases. Both the Livermore kernels (original implementation language: FORTRAN) and some WARP applications (original implementation language: W2) were translated to C, all other functions of the testsuite were readily available in C. It should be noted that the programming style of the Numerical Recipes in C is similar to FORTRAN coding style. We included a graph manipulation problem (*blocks*) that was very carefully handoptimized in C: pointers and address arithmetic were used excessively and the programmer

Input: *Parse tree, live ranges and conflict graph of a well structured program*

Output: *prohibited blocks, a set of basic blocks that may not be used for the computation of clique separators and a simplified conflict graph*

Method:

```
prohibited blocks := ∅;
for all nodes n in the parse tree in post order do {

        if n is a conditional {

                compute LOEN, LOEX, BGLOBAL, BLOCAL
                determine conditional continuous live ranges;
                eliminate holes from conditional broken live ranges;
                prohibited blocks += uneliminated holes;

                if conditional can not be linearized or collapsed {
                        prohibited blocks += blocks that form conditional;
                }
        } else {
                determine loop continuous live ranges;
                eliminate holes from loop broken live ranges;
                prohibited blocks += uneliminated holes;
        }
}
```

Figure 7-2: Outline of Phase 2

Input: *Simplified flow graph and prohibited blocks*
for each basic block **b** *s*(**b**), *the set of forward successors of* **b**

Output: *A partitioning of the register conflict graph*

Method:

```
for all basic blocks b ∉ {prohibited blocks} do {
    $separator(b) = \cup(IN(s)), s \in s(b)$;

}
partition graph based on separator cliques and partial graph sizes;
```

Figure 7-3: Outline of Phase 3

determined which variables were assigned to registers. We included this function to assess the shape of register conflict graphs of applications that are programmed in "typical" C style.

We chose the Livermore loops because they represent a standard benchmark of kernels that are used in many scientific applications [Pat/Hen 90]. While the Livermore kernels consist only of a collection of small kernels, we chose them because they are used by a large number of scientific applications. The same is true for the Numerical Recipes collection, and our selection of kernels from the Numerical Recipes collection was guided by their complexity: we chose the largest procedures from that collection of programs.

The set of applications from the WARP library consist of applications that consist of "real" programs instead of just kernels, and are (or were) in use on the WARP and IWARP machines. Both the Livermore loops and the examples from the WARP library represent scientific programs that are well suited for a globally optimizing compiler. Our graph manipulation program was chosen as an example of a non-numerical program, and in addition was carefully hand optimized. Such programs are usually hard to optimize by a compiler.

The histograms depicted in Figures 7-4 through 7-12 summarize some characteristics of our benchmark kernels. Figures 7-4, 7-5 and 7-6 depict the number of basic blocks, the number of live ranges and the nesting depth of the flow graph for the Livermore kernels. The same information for the numerical recipes is shown in Figures 7-7, 7-8 and 7-9, and for the WARP examples in Figures 7-10, 7-11 and 7-12.

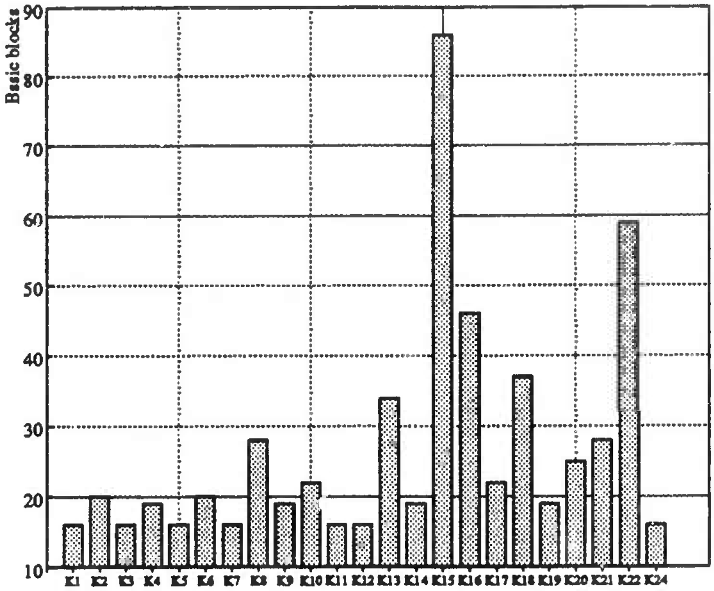


Figure 7-4: Livermore loops: basic blocks

For most benchmark functions, the number of live ranges is higher than the number of basic blocks. One example for which the number of live ranges is significantly smaller than the number of basic blocks is *block*, a graph manipulation program, very carefully hand optimized in C. The program contains 35 global variables, while the total number of live ranges is only 46. Because the number of basic blocks in that program is so much larger (126), it can be concluded that the "lack" of live ranges is due to the absence of temporaries after global optimizations. This is not surprising because hand optimized C programs that contain many pointers make it hard for a global optimizer to identify common subexpressions.

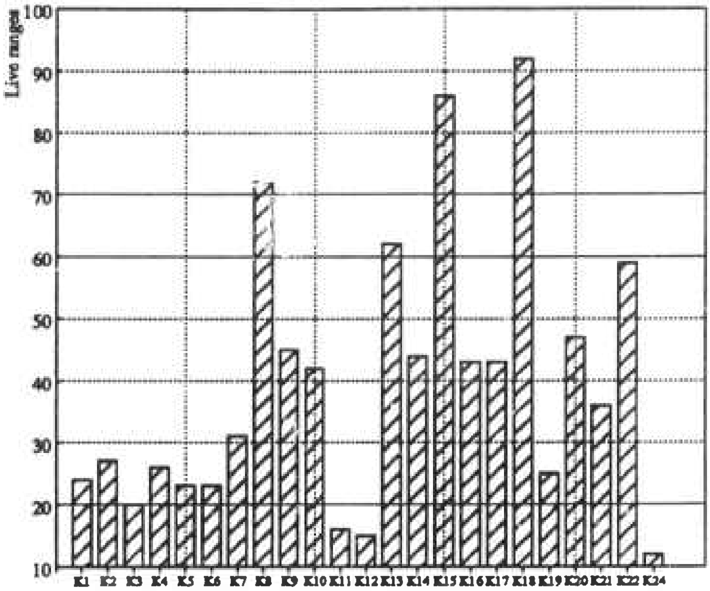


Figure 7-5: Livermore loops: number of live ranges

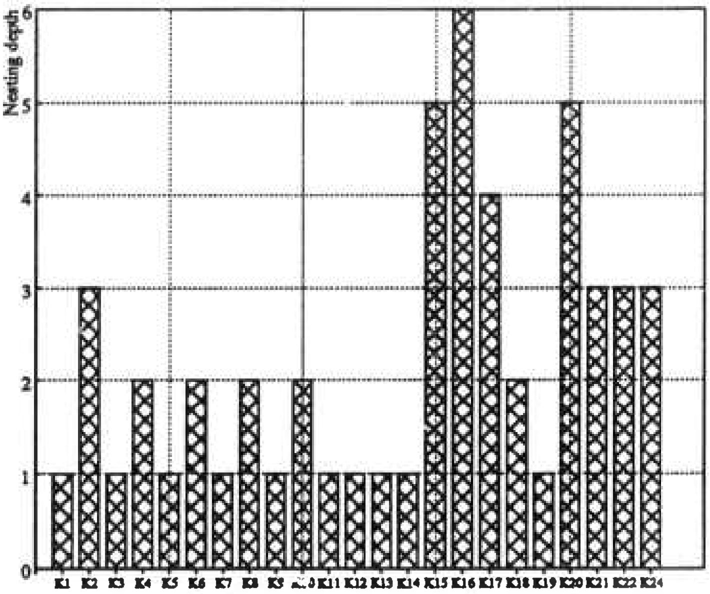


Figure 7-6: Livermore loops: nesting depth

7.3. Mapping register conflict graphs to interval graphs: evaluating Phase 2

Phase 2 of our method consists of applying transformations to a flow graph and a register conflict graph such that the original register conflict graph can be mapped to an equivalent interval graph. Mapping a register conflict graph to an equivalent interval register conflict graph can only be achieved if the following conditions are met:

1. All holes in broken live ranges can be padded,
2. All conditionals can be either collapsed or linearized.

We showed in Chapter 6 that we are able to pad holes optimally in polynomial time if they contain breaks,

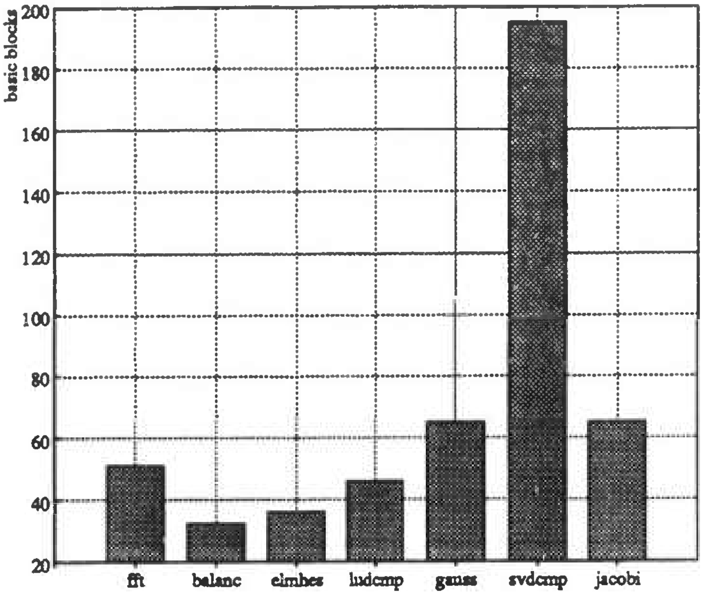


Figure 7-7: Numerical recipes: basic blocks

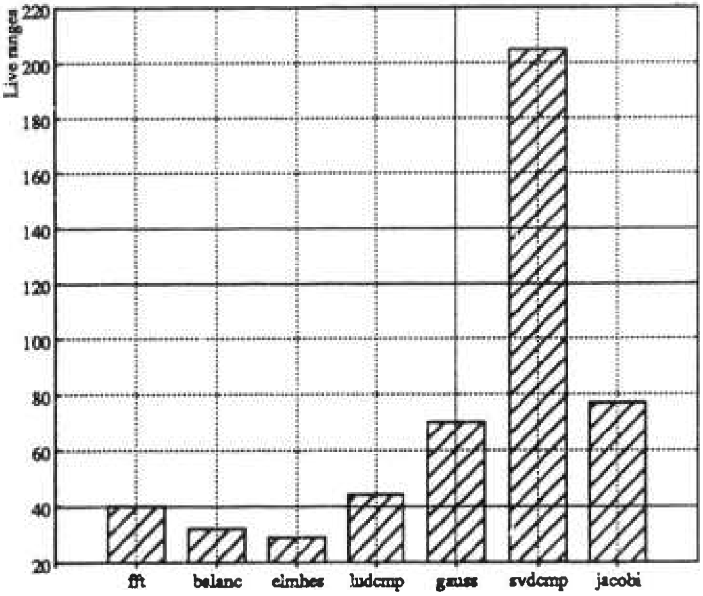


Figure 7-8: Numerical recipes: number of live ranges

or if holes can be padded entirely with perfect or imperfect matches. As a "reward" for successful node merging of all broken live ranges in a loop, the backarc of that loop can be eliminated from the flow graph without affecting the register conflict graph. Similarly, conditionals with only one nonempty branch clause can be simplified by removing the link from the split node to the join node if all holes that occur in the conditional can be eliminated. Therefore the simplification of conditionals with only one nonempty branch clause is equivalent to successful hole elimination.

Conditionals with multiple nonempty branch clauses - unlike those with only one nonempty branch clause - can cause non-interval register conflict graphs even if all live ranges are continuous. Thus, in our evaluation of the simplifications of conditionals, only those that contain at least two nonempty branch clauses must be considered.

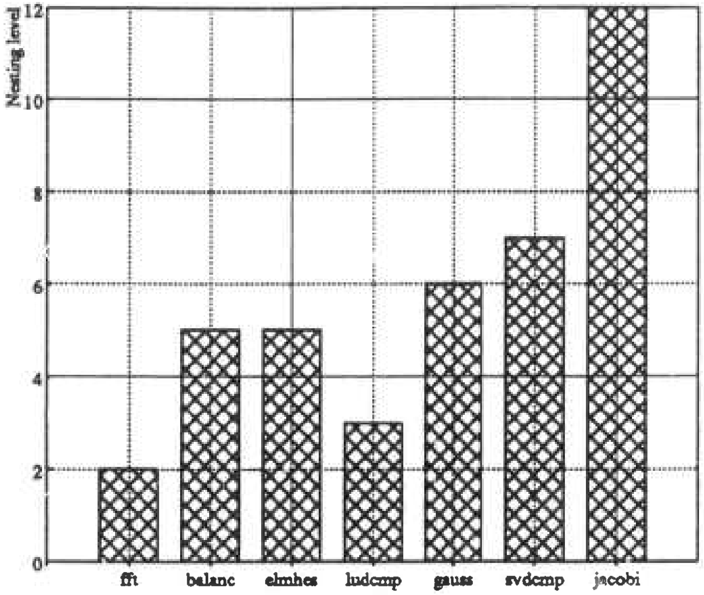


Figure 7-9: Numerical recipes: nesting depth

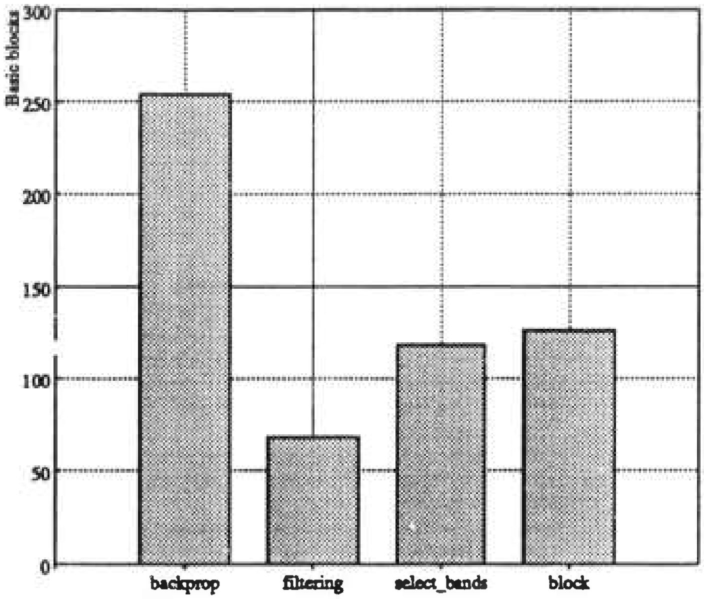


Figure 7-10: WARP examples: basic blocks

7.3.1. Holes and node merging in conflict graphs

Holes and node merging to pad holes is the focus of our first set of measurements. For each program, we counted the total number of holes in all live ranges and the successful attempts to pad those holes under one of our restrictions. The results for our benchmark kernels whose conflict graphs contained broken live ranges are summarized in Table 7-1.

The column labeled *holes* contains the total number of holes of all live ranges in the register conflict graph. The column labeled *padded* contains the number of holes that could be eliminated without node

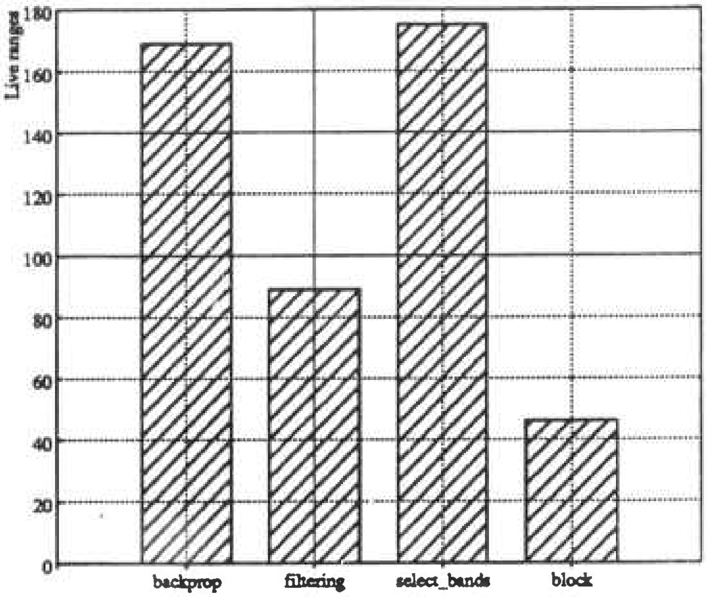


Figure 7-11: WARP examples: number of live ranges

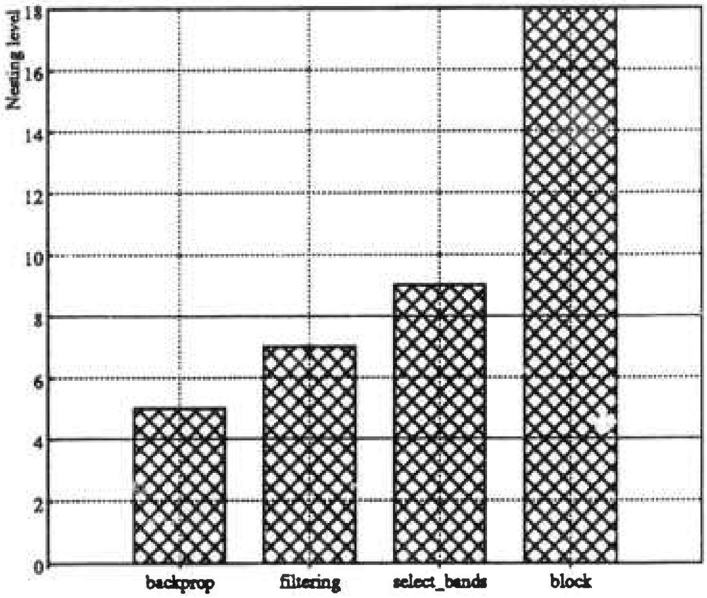


Figure 7-12: WARP examples: nesting depth

	Holes	Padded	Merged	Remaining
K20	1	1	-	none
ludcmp	1	1	-	none
svdcmp	20	9	9	2
backprop	2	2	-	none
block	2	2	-	none

Table 7-1: Node merging to eliminate holes

merging because no fits existed for these holes and the column labeled *merged* depicts the number of holes that were eliminated via node merging. The last column labeled *remaining* contains the number of holes that could not be eliminated by our method.

7.3.2. Discussion of the results for hole elimination

Of our 33 benchmark kernels, 5 had register conflict graphs with broken live ranges. Of the Livermore loops, only the register conflict graph of Kernel 20 contained a broken live range with one single hole that could be padded without node merging. Among the other programs, a few examples of the Numerical Recipes collection and of the WARP library contained broken live ranges. For all functions except *svdcmp* (single value decomposition), the holes could be padded without node merging. Single value decomposition is a fairly complicated example of the Numerical Recipe collection. The number of broken live ranges is 11, and the total number of holes is 20, 9 of which could be padded without node merging, and 9 of which could be eliminated via node merging. The large number of broken live ranges in *svdcmp* is caused by frequent re-definitions of variables that occur in conditionals. Our results indicate that for small kernels such as the Livermore loops, broken live ranges do not pose great problems. More complex programs such as the WARP and Numerical Recipes examples can contain holes, and in most cases our methods to pad holes with or without node merging are successful. In our examples, only two holes could not be padded with our method, and both occurred in *svdcmp*.

7.3.3. Sequentializing and collapsing conditionals

Our second set of measurements was conducted to examine how frequently conditionals could be linearized or collapsed. It was shown earlier that conditionals that consist of only one nonempty branch clause can be treated like straight line code provided all live ranges that occur in such conditionals are continuous. Hence, successful hole elimination is sufficient to find equivalent straight line code for the purpose of register allocation for such conditionals. Only conditionals with multiple nonempty branch clauses must be analyzed for simplification by sequentializing or collapsing.

The histograms shown in Figures 7-13, 7-14 and 7-15 summarize our results for sequentializing and collapsing conditionals. Each benchmark kernel that contained conditionals with at least 2 non-empty branch clauses is described by two bars. The left bar (shaded in grey) depicts the number of conditionals with at least 2 nonempty branch clauses. The right bar consists of a bottom and a top part.

The bottom part depicts the number of conditionals that could be linearized because they contained only *BLOCAL* live ranges or because *BLOCAL* live ranges were absent and all *LOEN* and *LOEX* live ranges formed cliques.

The top part depicts the number of conditionals that could be collapsed into just one path through the conditional because all *BLOCAL* live ranges of all branch clauses were E_1 or E_2 related to *BLOCAL* live ranges in a different branch clause. Conditionals that are collapsed must contain a combination of *LOEN* and/or *LOEX* live ranges with *BLOCAL* live ranges. If only *one* branch clause contains *BLOCAL* live ranges, no node merging takes place. If both branch clauses contain *BLOCAL* live ranges, the conditional can only be collapsed if the *BLOCAL* live ranges of both clauses are E_1 or E_2 related and have been merged in the conflict graph. E_2 related merging occurred in two of our examples: Livermore kernels 15 (2 merge operations) and 17 (1 merge operation). In all other cases, only one of the branch clauses contained *BLOCAL* live ranges and no node merging took place in the conflict graph.

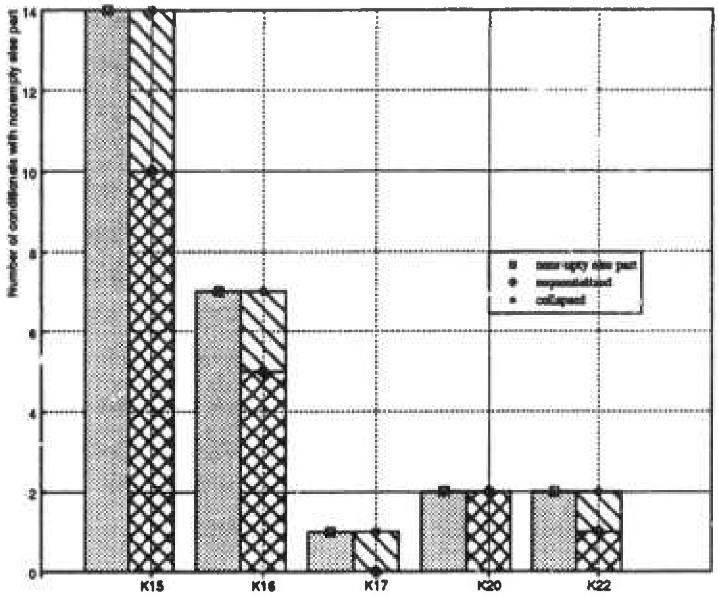


Figure 7-13: Simplification of conditionals in Livermore loops

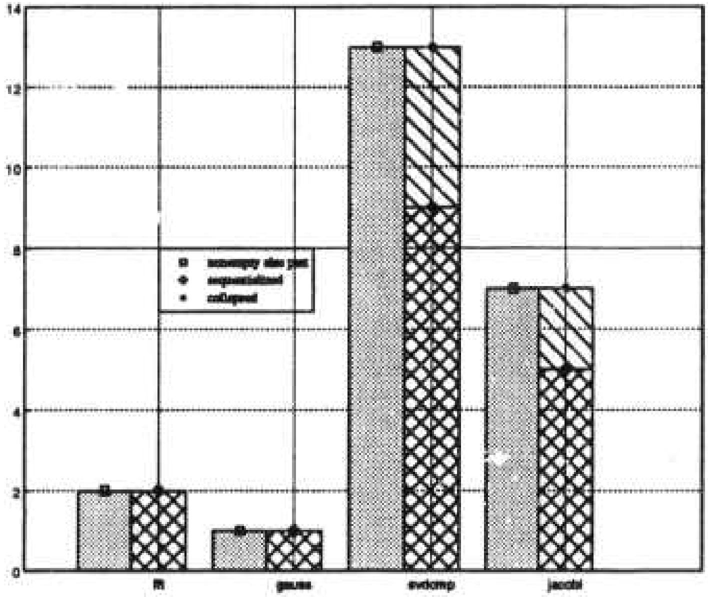


Figure 7-14: Simplification of conditiocals in Numerical Recipes examples

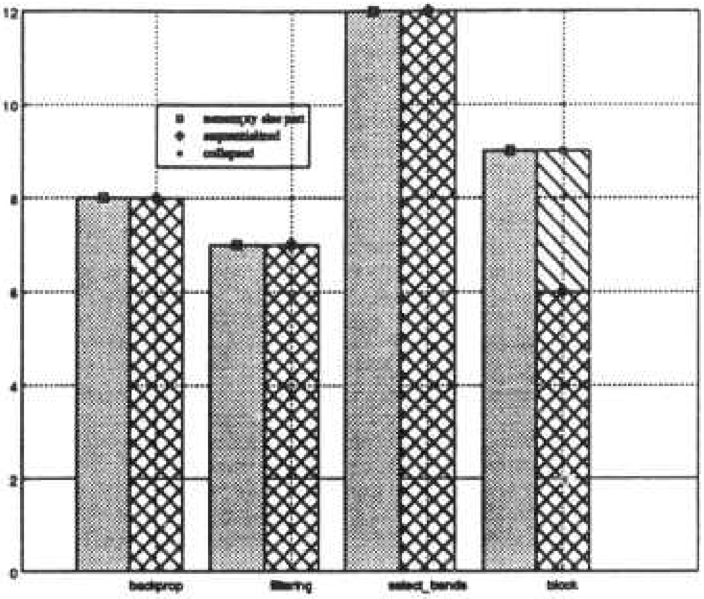


Figure 7-15: Simplification of conditionals in WARP and graph programs

7.3.4. Discussion of the results for conditional simplification

Every conditional in the programs of our testsuite could be linearized or collapsed. The number of linearized conditionals is much greater than the number of collapsed conditionals. In most cases, this was due to the absence of *BLOCAL* live ranges. For those conditionals that were collapsed, node merging occurred only in two cases (Livermore kernels 15 and 17). The number of merge operations was small compared to the total number of live ranges in these programs. Interestingly, outer conditionals whose branch clauses consist of complicated nested structures were always sequentializable. Typically, large conditionals that are sequentializable contain no *LOEN* or *LOEX* live ranges, so the *BLOCAL* live ranges in each branch clause are independent. In both cases that required node merging to collapse conditionals, one branch clause consisted of a long straight line structure while the second branch clause consisted of a short piece of straight line code consisting of only one basic block.

7.3.5. Combining the results for graph transformations

We conclude the first part of our evaluation by summarizing the overall result of our transformations. Of the 33 functions in our benchmark, 24 had interval conflict graphs and no node in those conflict graphs was altered (by adding to the set of basic blocks that form a live range) or merged. The conflict graphs of 9 benchmark kernels were transformed such that the resulting conflict graph was an interval graph. Such transformations included padding of holes and eliminating basic blocks from live ranges because branch clauses disappeared due to collapsing. There was only one function, *svdcmp*, whose conflict graph could not be transformed into an interval graph by our technique. The conflict graph contained 2 hot spots, one consisting of 4 and one of 13 live ranges. The remaining portions of the conflict graph (total number of original nodes: 205) were detected as interval graphs or transformed into interval graphs. The chromatic numbers of the conflict graphs of our benchmark kernels are summarized in Figures 7-16, 7-17 and 7-18.

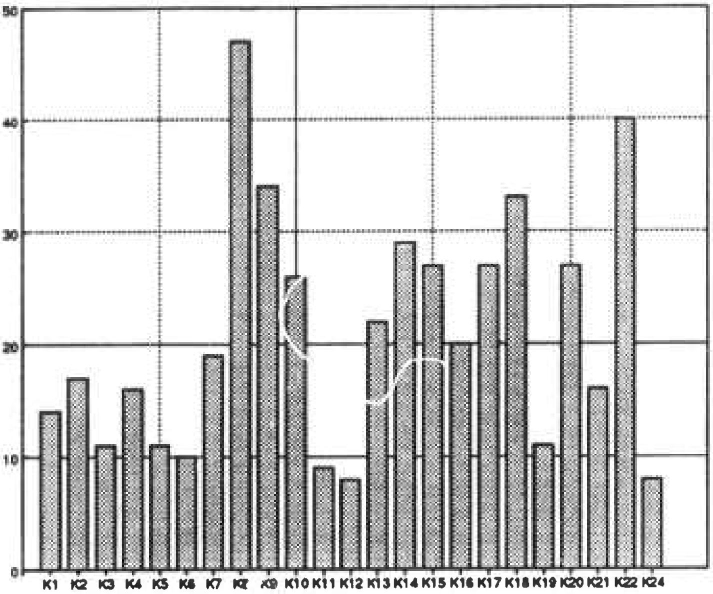


Figure 7-16: Chromatic numbers: Livermore loops

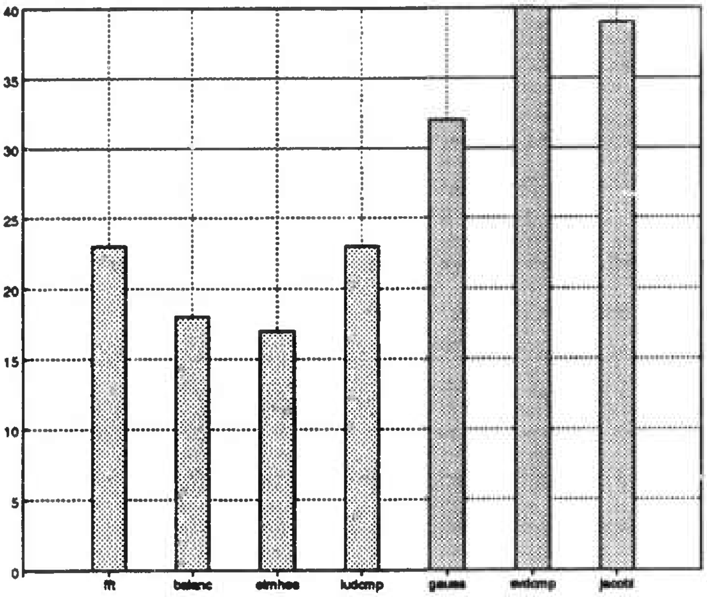


Figure 7-17: Chromatic numbers: Numerical Recipes

7.3.6. Discussion of graph transformation results

We have seen that in all but one cases the register conflict graphs of our programs could be simplified to interval register conflict graphs. The chromatic numbers of the graphs were largest for the more complicated examples in our testsuite. Single value decomposition (*svdcmp*) was the only program that had "hot spots" in the register conflict graph, that is there were two portions in the register conflict graph that could not be simplified to interval graphs for the purpose of global register allocation. The number of basic blocks in those portions (23) is small compared to the total number of basic blocks in that program (195). Further, the set of live ranges that fit into those portions was small, so exhaustive search for that

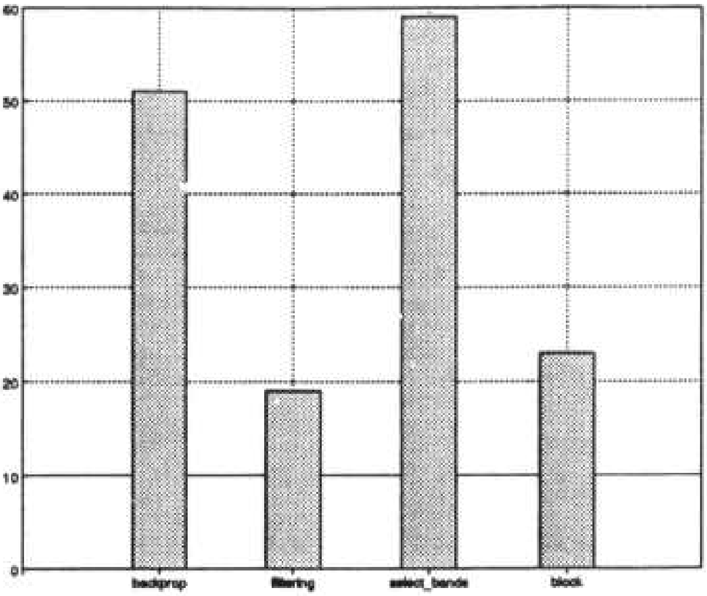


Figure 7-18: Chromatic numbers: WARP and graph programs

subgraph of the conflict graph is feasible. Interestingly, the chromatic number of one of the hot spots dominated the chromatic number of the clique separated components of the remaining conflict graph. So exhaustive search of a small component of the register conflict graph enabled us to optimize the overall chromatic number.

All except one conflict graph of our benchmark functions were mapped to an equivalent interval graph by our method. The one non-interval conflict graph contained only one small non-interval section, such that most of the conflict graph could be mapped to equivalent interval graphs. The number of clique separators found in each register conflict graph is bounded by the number of basic blocks that could be transformed into straight line code for the purpose of register allocation. For *svdcmp*, 85% of all basic blocks and for all other benchmark kernels, *all* basic blocks could be transformed to straight line code, so at least 85% of all clique separators were located for *svdcmp*. For all other kernels, our method was able to locate *all* clique separators. In the second part of this chapter we show how clique separators are used to partition a register conflict graph into independent components.

7.3.7. Parallelization based on clique separators

The parallelization of global register allocation is useful if optimal global register allocation is expensive. Given k registers and a k-colorable interval register conflict graph, an optimal assignment of registers can be found in linear time. In that case, significant parallel speedup for global register allocation can not be expected because the overhead of parallel task management and re-combination of individual results would cancel any speedup over sequential register allocation.

Optimal global register allocation is expensive if the register conflict graph is not an interval graph and if the register conflict graph is not k-colorable, and live ranges must be split or spilled to memory. By our analysis, the non-interval graph portion of practical register conflict graphs can be restricted to relatively small subgraphs.

Arbitrary register conflict graphs

Exponential coloring algorithms are feared for a reason: suppose we have an upper bound of k colors needed to color an arbitrary graph with 15 nodes, but would like to find an optimal coloring, possibly reducing the number of registers. In this scenario, exhaustive search requires inspecting k^{15} different combinations - for $k=5$, the number of combinations is several billions. In reality, exhaustive search need not be so prohibitively expensive, because large quantities of combinations can be pruned from the search space. It is very difficult to make an accurate prediction of the time it takes to produce an optimal coloring of an arbitrary graph.

Spilling

If the chromatic number of an arbitrary or interval register conflict graph exceeds the number of available registers, spilling and/or live range splitting must be applied. Good spilling is crucial to the execution time of a program, and optimal spilling is NP complete even for straight line code, and therefore for interval register conflict graphs. Because good spilling decisions can be expensive, a parallelization based on clique separators is likely to achieve good parallel speedup. It is a disadvantage that the optimality of spilling decisions for individual portions of the register conflict graph can not always be maintained when clique connected components are combined.

It is difficult to predict the complexity of global register allocation in the presence of spilling or non-interval register conflict graphs. The reason is that heuristics for good spilling or good colorings of non-interval portions can be arbitrarily expensive. In our benchmark the chromatic numbers of the register conflict graphs range between 8 and 59. The number of registers in most current computer architectures is insufficient for a number of our conflict graphs. Depending on the spilling heuristics employed, the running time of global register allocation can be large for the register conflict graphs of our benchmark.

7.3.8. Method of parallelization: an example

The parallelization of global register allocation via clique separators is best explained by example. Figure 7-19 depicts an interval register conflict graph and its partition into two subgraphs via a clique separator. The graph labeled "conflict graph" depicts the conflict graph of the entire program that consists of basic blocks $\{1,2,3,4,5,6,7,8,9,10\}$. The set of live ranges consists of $\{a,b,c,d,e,f\}$. The chromatic number of the conflict graph is 3. Because the program consists of straight line code, every basic block can be chosen to find a separator clique. The box drawn around basic block 5 and across the live ranges that contain block 5 depicts the separator clique based on 5. Block 5 is contained by live ranges $\{b,c,d\}$. Therefore, $\{b,c,d\}$ form a clique that partitions the conflict graph into two independent components, the top and bottom part depicted below the original conflict graph. Both the top part and the bottom part can be colored independently, as long as the separator clique is colored in both parts. The colorings of both parts can be combined by adjusting the colors used for the live ranges in the separator clique, namely $\{b,c,d\}$. The number of live ranges whose coloring must be adjusted depends on the number of live ranges colored with the colors used for b,c and d.

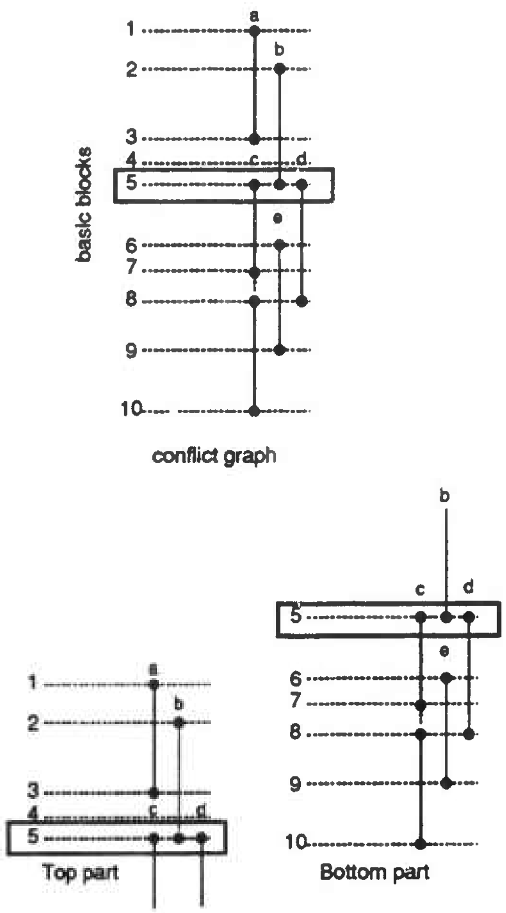


Figure 7-19: Parallelization via clique separators

Note that in a parallel implementation based on clique separators, the nodes of each separator clique must be colored twice, which is not necessary in a sequential implementation. So the parallel algorithm does more "work" than the sequential algorithm. In our example, the size of the separator clique is 3. Given two parallel processors, 3 extra coloring steps must be performed plus the re-naming during post processing. Because the size of the separator clique is equal to the chromatic number of the conflict graph, *every* live range in the conflict graph must be re-colored, if none of the colors for the elements of the separator clique match. Had we chosen a separator clique that is smaller, both the redundant work and the number of re-coloring steps would be smaller, depicted in Figure 7-20. The conflict graph is identical to that depicted in the previous figure, but this time the separator is chosen based on block 4. The new separator clique consists only of $\{b\}$, and given an independent coloring for both parts separated by $\{b\}$, at most one live range must be re-named when the two colorings are combined.

This example demonstrates that the size of a separator clique determines both the amount of redundant work spent in parallel register allocation and the post-processing time. In the nest section we present measurements of separator cliques for some of our benchmark kernels.

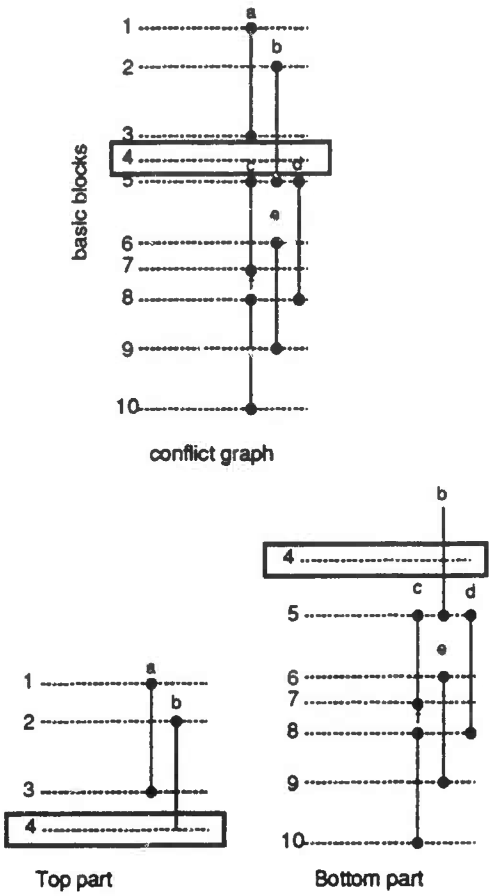


Figure 7-20: Chosing a small separator clique

7.4. Parallelization based on clique separators: evaluation of Phase 3

Our method transforms a flow graph to a straight line sequence of basic blocks that are equivalent for the purpose of register allocation. By Theorem 11 of Chapter 5, this sequence of basic blocks can be used to compute an ordered sequence of cliques in the conflict graph, each of which is also a clique separator. The parallelization consists of partitioning the register conflict graph via these clique separators.

In interval conflict graphs, the processing time for the parallel tasks is correlated to the number of live ranges in each subtask. So if it is possible to divide the conflict graphs such that the clique connected components contain an equal number of nodes, and in addition the clique separators at those points are not too large, parallelism based on clique separators is intuitively successful. We present measurements of both the separator clique sequences and the distribution of live ranges for some examples of our benchmark kernels in Figures 7-21, 7-22, 7-23, 7-24, 7-25, 7-26 and 7-27. The x-axis depicts the basic blocks in the flow graph, the y-axis depicts the number of live ranges as a percentage of the total number of live ranges in the conflict graph. For each basic block that is not marked *prohibited* by our method, the bottom line depicts the size of the separator cliques as percentage of the total number of live ranges, and the top line depicts the accumulative number of live ranges. If for example the value of the bottom line for basic block 10 is 15, the value of the top line 45, then the size of the separator clique is 15% of the total number of live ranges and 45% of all live ranges start at block 10 or earlier.

The results for *svdcmp* are depicted in Figure 7-21. The total set of live ranges is distributed relatively evenly over the set of straight line basic blocks. At every basic block, the size of the separator cliques is only a small percentage of the total number of live ranges. Between individual separator cliques, there are relatively large differences. Note that at three basic blocks, 67, 92 and 124 the size of the separator clique is very small. These basic blocks coincide with loop exits in the program. Basic blocks for which the size of the separator clique is not shown are the prohibited blocks among which our method is unable to find separator cliques.

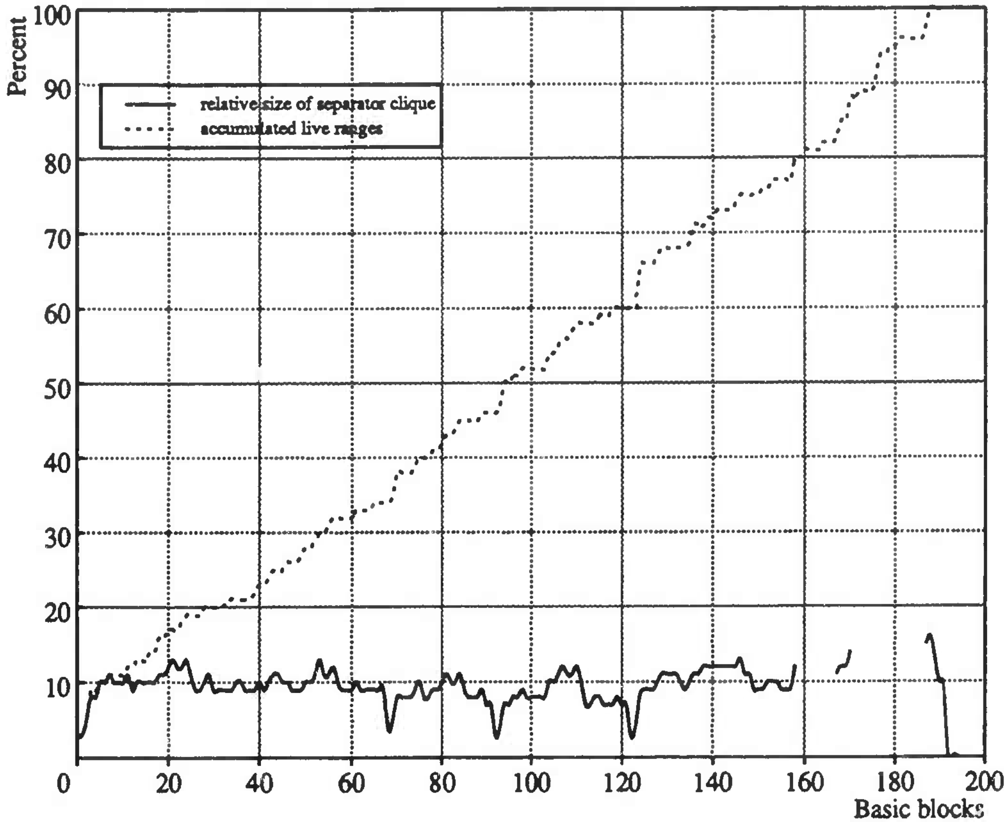


Figure 7-21: Separator cliques and accumulated live ranges for svdcmp

We demonstrate the use of the information depicted in Figure 7-21 for the parallelization of global register allocation by an example. Suppose we wish to parallelize global register allocation for *svdcmp* with 3 processors, such that each processor processes an equally large component of the conflict graph. The live ranges must be partitioned into 3 sets, each consisting of 33% of the conflict graph. The clique that separates the first 33% of the conflict graph is found at basic block 60, the next component consists of all live ranges between basic blocks 60 and 130 (where 66% of the live ranges have been accumulated) and the last component consists of the live ranges between basic blocks 130 and 195. Obviously only basic blocks for which the separator clique exists (i.e. non-prohibited blocks found in Phase 2 of our algorithm) can be used to partition the live ranges.

Figure 7-22 shows the same data for the backpropagation algorithm of the WARP library. The line depicting the accumulated live ranges rises very steeply for the first few basic blocks, and then climbs almost linearly. This is due to a relatively large number of global variables in that program. The size of the separator cliques is very even throughout the program, but compared to the total number of live ranges noticeably higher than for *svdcmp*. Again, this can be explained with the number of global variables that remain live throughout the program, so they are part of every separator clique. The code for that program is typical "spaghetti code", a long sequence of not too complex programming constructs.

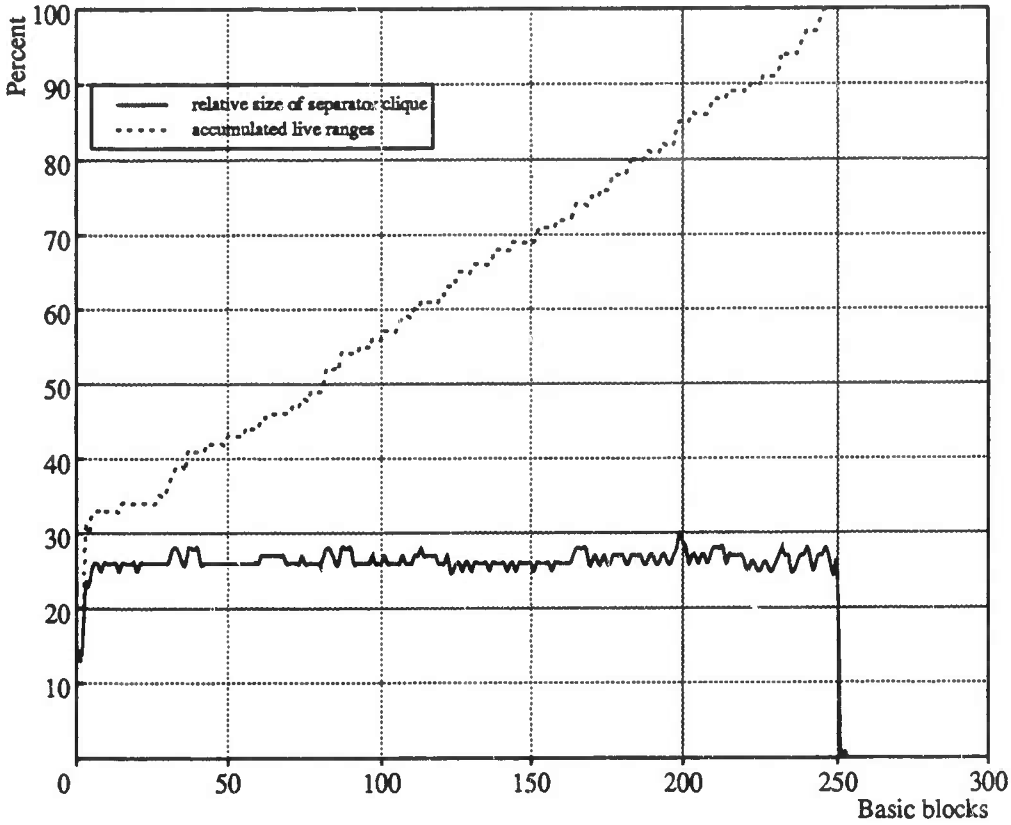


Figure 7-22: Separator cliques and accumulated live ranges for back

Our next example is the frequency domain filtering routine, depicted in Figure 7-23. The size of the separator cliques varies greatly throughout the program, and remains high for the last set of basic blocks that follow block 48. The variations in the separator cliques can be explained with the structural complexity of the program. Note that around basic block 48, almost 100% of the live ranges have been accumulated. The persistently large separator cliques for the same portion of the program indicate computation involving a large set of variables. The frequency of small separator cliques in the first part of the program together with a relatively linear increase of the accumulated live ranges indicate that the conflict graph can be parallelized successfully.

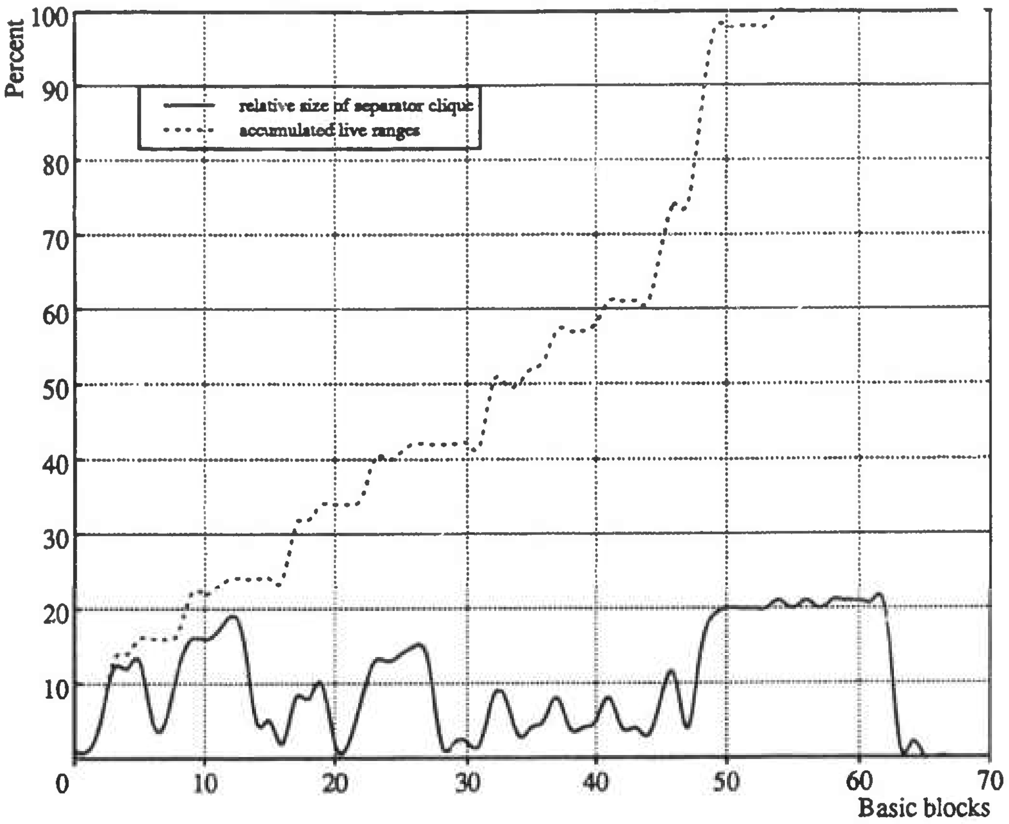


Figure 7-23: Separator cliques and accumulated live ranges for filtering

The *filtering* routine is inlined in the *select* kernel shown in Figure 7-24. This is reflected in the shape of both the accumulated live range curve and the separator clique curve: between basic blocks 20 and 80, the shapes of both curves are almost identical to the curves of Figure 7-23. Note that overall the size of the separator cliques is much larger in relation to the total number of live ranges than for the frequency domain filtering routine in isolation. This is due to variables local to select, that remain live throughout the inlined code for frequency domain filtering.

Our next example is the *jacobi* kernel in Figure 7-25. The size of the separator cliques is very irregular, again caused by the structural complexity of the program. In the second half of the flow graph, starting at basic block 31, the size of the separator clique is very large in relation to the total number of live ranges. Small cliques occur only at the beginning of the program. This example shows the tradeoff between finding parallel tasks of equal size and the size of the separator cliques. If the size of the separator cliques is the main factor for the parallelization, the size of the parallel tasks must be very uneven - on the other hand equally sized parallel tasks can only be obtained by partitioning via large separator cliques.

As our last two programs we picked the two most complex Livermore kernels, Kernel 15 and Kernel 22. The data for Kernel 15 is shown in Figure 7-26, and the structure of the code is reflected in the size of the separator cliques: the kernel consists of two separate loops, each of which performs computations on variables local to each loop. The size of the separator cliques is large in the basic blocks that occur inside

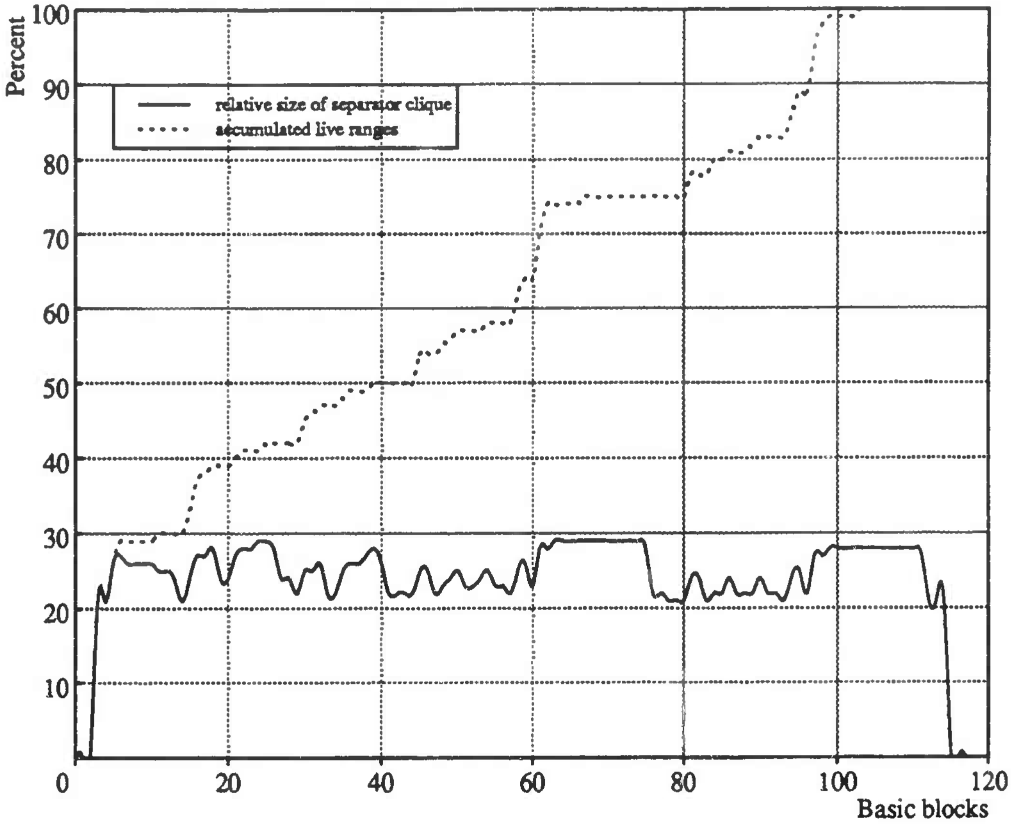


Figure 7-24: Separator cliques and accumulated live ranges for select

the loops, and the separator clique of basic block 49 that partitions the first loop from the second is extremely small. This indicates that the register conflict graph can be partitioned into two almost completely independent parts, each of which contains about 50% of the live ranges.

The data for Livermore Kernel 22 is depicted in Figure 7-27. The shapes of both the accumulated live ranges curve and the separator clique curve are typical for single loops: almost all live ranges are accumulated within the first few basic blocks, and almost all variables live throughout the loop, indicated by the large size of the separator cliques for the basic blocks that form the loop body, blocks 5 through 53. Because the bulk of live ranges start in the same basic block (note the steep ascend of the accumulated live range curve at basic block 5), it is not possible to partition the conflict graph into two subgraphs that contain equally many nodes - in addition to very large separator cliques. Hence, the parallelization of global register allocation for such graphs is useless.

7.4.1. Tradeoffs between parallelization strategies

The conflict graph of our single value decomposition program (Figure 7-21) has properties that make it easy to partition. First, the accumulated live range curve rises almost linearly with increasing basic block numbers. This means that it is easy to partition the register conflict graph into chunks of even size. The size of the separator cliques is fairly even, and, more importantly, a relatively small fraction (about 10%) of

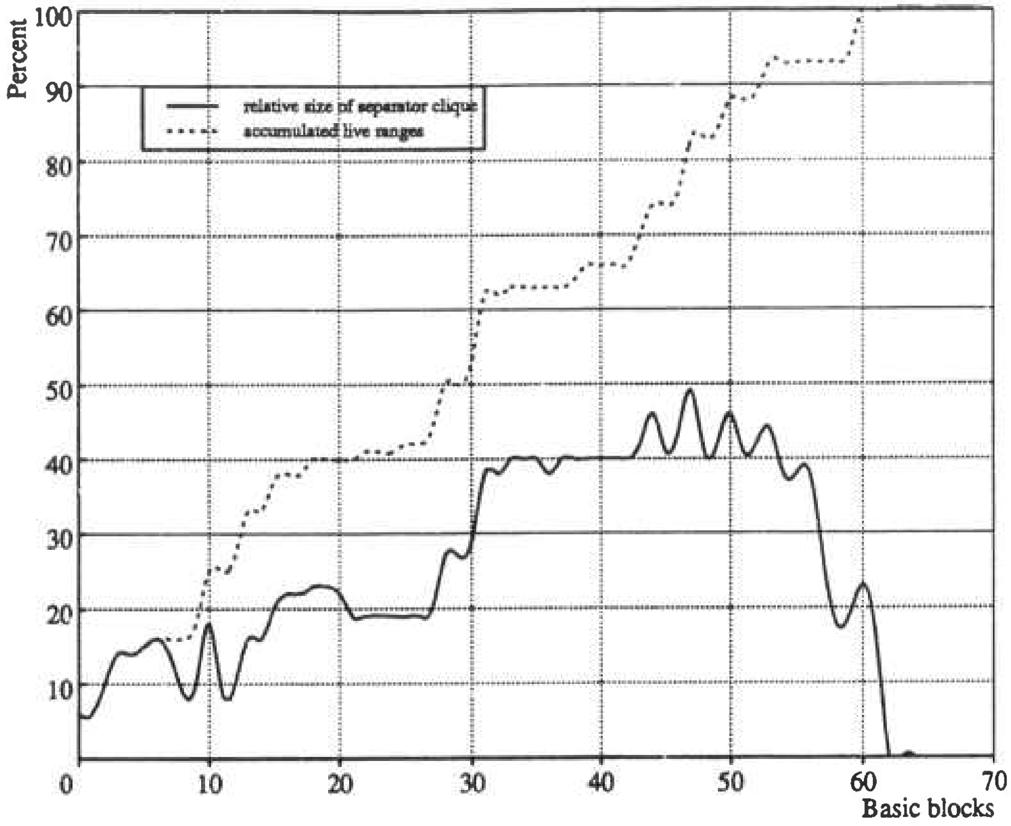


Figure 7-25: Separator cliques and accumulated live ranges for jacobi

the total number of live ranges. Using 10 pro:essors to process this conflict graph in parallel means that the size of the separator clique is equal to the number of nodes in the partial conflict graphs; hence the number of processors that can be used effectively is smaller than 10.

Even though the conflict graph of our backpropagation program has similar characteristics, the size of the separator cliques is much larger in relation to the total number of live ranges (ca. 25%). Distributing global register allocation for this kernel across 4 processors means that for each component of the conflict graph the size of the separator clique is equal to the number of nodes - so only fewer than 4 processors can be used effectively in parallel. The large size of the separator cliques is a consequence of a large number of global variables that remain live throughout the program.

Our frequency domain filtering program produces a conflict graph in which the varying sizes of the separator cliques can not be ignored during the partioning. About 40% of the separator cliques are very small, whereas the other 60% of the separator cliques are very large compared to the total number of nodes in the conflict graph. A simple partition based merely on graph size might not yield good speedup. This phenomenon is ta[illegible]n to the extreme in the example depicted in Figure 7-26 - only a few clique separators are small enough to make parallelization of global register allocation worthwhile. Only two clique separators (at basic blocks 49,50) partition the conflict graph into large enough chunks so that the maximal task size becomes sufficiently small.

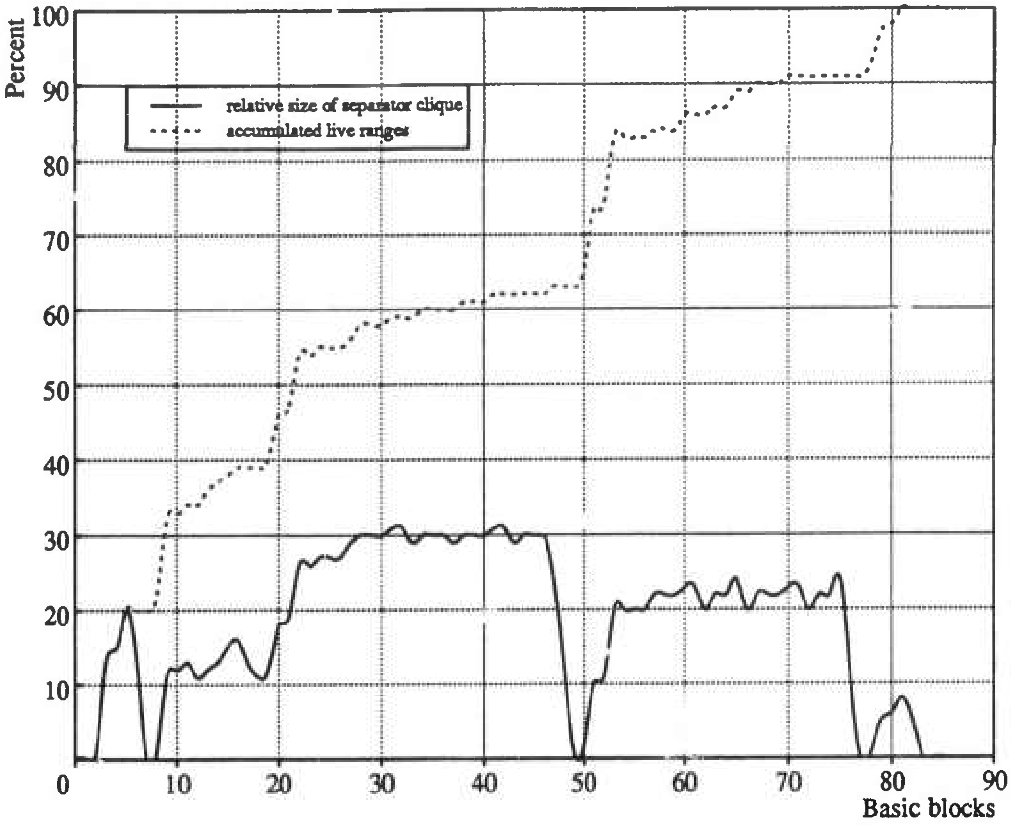


Figure 7-26: Separator cliques and accumulated live ranges for Livermore Kernel 15

To summarize our observations, good parallel speedup can only be expected when the size of the separator clique is significantly smaller than the number of nodes in the partial conflict graphs. Our measurements indicate that for small kernels this is hardly achieved. Our single value decomposition kernel is one of the longest kernels in our testsuite, and the number of global variables is small. The relation of the separator clique size to the total number of live ranges is favorable, and we conclude that for longer programs, more parallelism can be found in global register allocation.

While procedure inlining increases the size of individual functions, our experience is that the size of the separator clique grows with the context of the inlining functions: live ranges of the inlining functions co-exist with the live ranges of the inlined function, thereby increasing the size of the separator clique. One possibility to overcome this problem is to reduce the size of the separator clique by splitting live ranges of the inlining procedure.

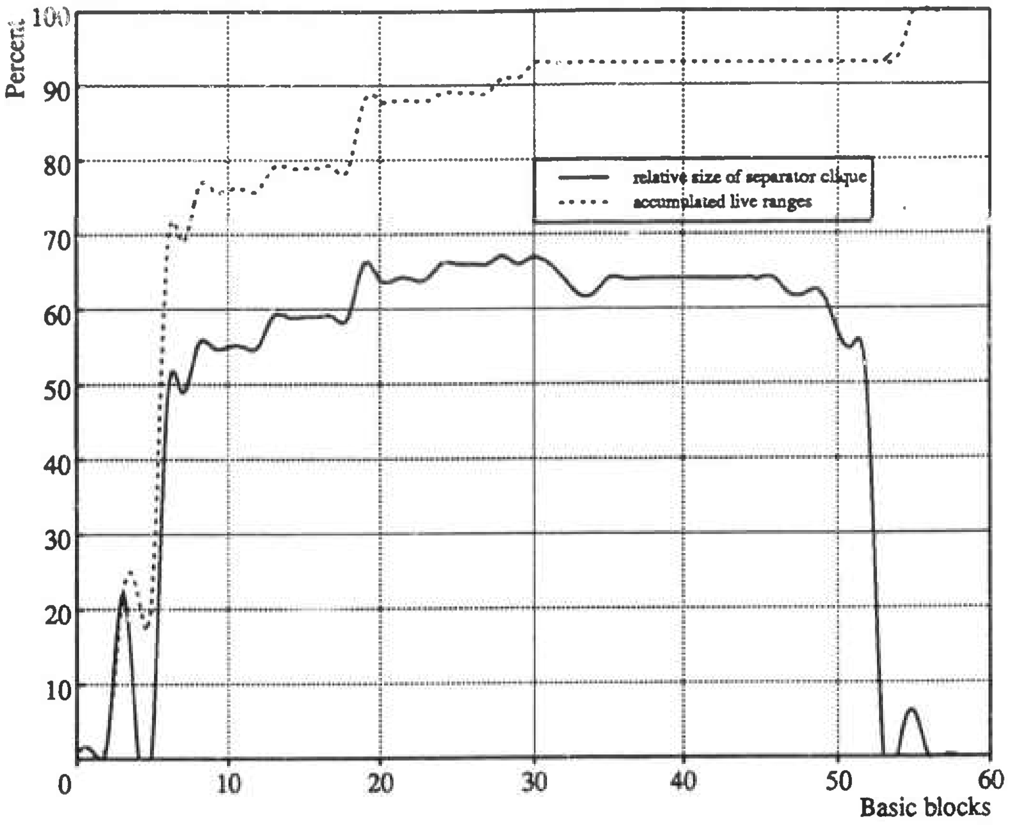


Figure 7-27: Separator cliques and accumulated live ranges for Livermore Kernel 22

7.5. Related work

Global register allocation is based on the conflict graph that denotes the register conflicts of the *entire* program. In some implementations of global register allocation via graph coloring, the entire register conflict graph is built before the coloring process starts. By assigning priorities to variables based on where in the program they occur, the program structure implicitly guides the order in which variables are assigned to registers. Different implementations differ mainly in the heuristics that are used to decide which variables to spill to memory if there are not enough available registers [ChowHen 90, Briggs et al 89, Bernstein et al 89].

Other approaches to global register allocation use program structure more explicitly. In the research reported in [GupSofSte 89], the register conflict graph is built and colored separately for each path through the flow graph. The goal is to build and color a register conflict graph incrementally, motivated by space problems when the entire conflict graph is kept in memory. Since only one straight line path through the flow graph is considered at a time, the register conflict graph is assumed to be an interval graph and hence clique separators are used for data partitioning. When the colorings of conflict graphs of individual paths are combined, move instructions are used to adjust colorings. We see two problems with this approach: First, even though straight line paths are considered individually, the presence of broken live ranges does not ensure that the conflict graphs for these paths are really interval graphs. As a result, a clique found in the register conflict graph of one path might not even be a separator clique for the conflict graph of one individual path through the flow graph. Second, when the colorings of the paths through a conditional are merged, there is no mechanism to enforce that *BGLOBAL* or *LOEX* live ranges are colored identically and colorings must be adjusted by introducing unnecessary spill and move instructions.

A strictly hierarchical approach to global register allocation via graph coloring is reported in [CallahanKoblenz 91]: register conflict graphs are built and colored based on the parse tree of the program. The motivation is to concentrate on the register conflict graphs of critical portions of the input program, for example innermost loops. An overall coloring is derived by combining individual colorings; colorings that do not match are adjusted by spill and move instructions that can be avoided with the method presented in this thesis. The trace scheduling compiler described in [FreuRut r)] uses a similar approach, though register allocation is performed hand in hand with the code scheduler.

Common to all approaches is their heuristic nature. If parts of the register conflict graph are colored separately, the overall coloring is derived by inserting cleanup operations like move instructions if the two individually colored pieces do not fit together Even if individual pieces can be colored optimally optimality of the overall coloring is in general not preserved when colorings are combined.

7.5.1. Chapter summary

Our method was very successful for our benchmark kernels. Every conditional could be transformed into equivalent straight line code, in most cases without node merging. Broken live ranges occurred in several programs, and the number of broken live ranges was related to the structural complexity of the programs. In all but two cases (both in the same program), holes could be eliminated. Consequently, all except one kernel could be mapped to a straight line flow graph that was equivalent for the purpose of global register allocation, and the resulting conflict graph was an interval graph. Only for one kernel an interval register conflict graph could not be found. For that kernel, we identified a small subset of nodes in the register conflict graph that induced non-interval portions in the conflict graph.

Because it is difficult to predict the sequential execution time for a non-interval graph or for a graph in which nodes must be spilled to memory, the parallel speedup can not be assessed accurately. Our measurements of separator clique sizes demonstrates that our method finds enough clique separators such that a variety of strategies can be applied to parallelize global register allocation.

Chapter 8
Conclusions

Program structure plays an important role for the parallelization of global compiler optimizations. We distinguish between structured and unstructured compiler optimizations. In structured compiler optimizations, data locality is given by explicit program structure: results for non-nested programming constructs can be combined to an overall result without backtracking. In unstructured compiler optimizations, there is no easy way to partition the data into independent components. Data partitioning is based on characteristics of the underlying data structure, and program structure is used implicitly to deduce those characteristics.

In this thesis, we developed parallel frameworks for interval analysis and global register allocation, representing a structured and an unstructured optimization respectively. The thesis demonstrates that for both types of compiler optimizations, program structure plays a key role in finding a data partitioning.

8.1. Parallel interval analysis

In interval analysis, the basic blocks of a program flow graph are processed in an order that is given by the loop structure of the input program. This leads to a natural partitioning of the data for parallelization: the data flow information of loops that are not nested within each other can be computed independently and combined to form an overall result without backtracking. Our parallelization of interval analysis is based on a tree called the *complete interval tree* that captures the loop structure of the input program. Each node in that tree is a task in the parallel implementation of interval analysis, and all nodes in that tree that are not descendants or predecessors of each other can be processed in parallel.

The goal of our implementation of parallel interval analysis was to assess the number of parallel processors that can operate effectively for a given complete interval tree. When the execution time for each parallel task is known beforehand, the minimal time to process the complete interval tree is equal to the length of the longest path from the root of the tree to a leaf node. Given the minimal processing time for a complete interval tree T, we developed a lower bound for the minimal number of processors needed to process T in minimal time (finding the minimal number of processors is NP complete).

Our parallelization of interval analysis is extremely simple, and one drawback is that the data partitioning depends directly on the loop structure of the input program. Hence, the size of the parallel tasks is pre-determined by the loop structure, and there is no flexibility when the parallel tasks are of uneven sizes. In some cases this can lead to the lack of parallelism when a complete interval tree is processed.

Our implementation shows that in practice potential parallel speedup decreases when the number of processors used is too large; in all cases the number of processors that could be used effectively in practice was significantly smaller than the minimal number of processors needed to process the complete interval tree in minimal time. In other words, due to system overhead the theoretically optimal speedup could not be achieved. We conclude that finer grained parallelism in which more processors can be used in parallel does not result in better observed speedup, and that our simple approach to the parallelization of global data flow analysis works well in practice.

8.2. Global register allocation

While for parallel interval analysis the division of the problem in parallel tasks is explicitly given by the loop structure of the input program, the role of program structure in the parallelization of global register allocation is very different. In global register allocation, program structure is used to guide the *analysis* of the register conflict graph, which must be partitioned for parallel processing. Goal of this analysis is to locate clique separators in the conflict graph, and we achieve this by determining portions of the register conflict graph that are interval graphs.

We used the fact that register conflict graphs for straight line code are interval graphs, so in a well structured program the only source for non-interval portions of the register conflict graph are conditionals and loops. When certain restrictions are met by the live ranges that occur in conditionals or loops, we can infer that the corresponding portion of the register conflict graph is an interval graph. Even when those restrictions are not met by the live ranges of a loop or a conditional, there are situations in which we can *create* an interval register conflict graph via node merging.

To characterize live ranges that are responsible for non-interval register conflict graphs, we developed a structured model for global register allocation in which knowledge about program structure is encoded in the live ranges of the input program. Given this knowledge in a register conflict graph, the conflict graph is analyzed in a structured way, one loop or conditional at a time.

Our model provides a basis that permits us to improve register allocation, both for sequential and parallel implementations. The advantages of our method for sequential register allocation are as follows:

1. Interval portions of the register conflict graph are detected and can be colored optimally in linear time.
2. Clique separators can be found systematically in interval portions of the register conflict graph. Clique separated components of the register conflict graph can be colored independently and combined to an overall coloring in linear time without compromising the quality of the overall solution. Being able to partition a register conflict graph into small components means that the overall running time can be reduced.
3. All cliques in interval graphs can be enumerated. The size of the cliques in interval portions of register conflict graphs is known. Only live ranges that are members of cliques that are larger than the number of registers must be considered for spilling. In most cases, live ranges that must be considered for spilling are only a *subset* of all live ranges of the input program, and this subset is known *before* the coloring process. This is a definite advantage over standard methods for spilling in which *all* live ranges that are still part of the register conflict graph must be considered for spilling indiscriminately.

4. In general finding an optimal coloring for a non-interval portion of the register conflict graph is NP complete. Our measurements indicate that such portions usually consist of only small parts of a register conflict graph, so using expensive heuristics or even exhaustive search for the coloring such portions need not be prohibitively expensive. Non-interval portions of the register conflict graph are clique separated from the interval portions, so good (or optimal) colorings for non-interval portions can be incorporated into an overall coloring in linear time.

5. The knowledge about clique separators is useful for hierarchical register allocation: crucial portions of the register conflict graph (example: portions that correspond to innermost loops of the flow graph) can be colored separately and combined with the overall coloring in a systematic and structured way.

8.2.1. Evaluation of the model

Goal of our implementation was to evaluate how often restrictions that enable us to detect or create interval register conflict graphs are met by the live ranges of realistic programs. Our model is sufficient for the analysis of register conflict graphs when we are able to detect large interval portions and many clique separators. Our measurements show that for most input programs our method is able to detect interval register conflict graphs and therefore a large number of clique separators.

8.2.2. Parallel global register allocation

Given the knowledge about the location of clique separators in a register conflict graph, the parallelization of global register allocation is straightforward. A partitioning of a register conflict graph that allows good parallel speedup must have the following characteristics:

- The size of the graph portions that are processed in parallel must be equal.
- The size of the separator cliques must be small to decrease the re-combination cost.

Our results indicate that for small kernels, the size of the separator cliques is so large that only very few (2-4) processors could be used efficiently in parallel. For larger kernels the size of the separator cliques tended to be small in relation to the total number of nodes in the conflict graphs and therefore the number of processors that can be used in parallel effectively is a little larger. The abundance of separator cliques allows a variety of strategies for the parallelization.

8.3. Future work

We examined the correlation between program structure and global compiler optimizations under two different aspects. First, the explicit use of program structure for the *parallelization* of compiler optimizations that operate based on a program's loop structure. The effectiveness of this approach was shown for interval analysis, and the implementation of parallel interval analysis of only one data flow equation showed potential for speedup in practice. Second, the implicit use of program structure for the *analysis* of data structures processed in less structured optimizations which leads to a subsequent parallelization. By developing a structural model for global register allocation we showed that this concept works well.

8.3.1. Explicit use of program structure: extensions

Our work in parallel interval analysis can be deepened in two aspects. First, it is important to examine the correlation between program structure and the best possible speedup in detail. Ideally this work should lead to a simple model that permits us to predict the expected parallel speedup for a given input program. This is particularly important for practical implementations of parallel compilers: the automatic parallelization of such optimizations is only successful when the amount of parallelism and therefore the number of processors that can operate effectively in parallel can be predicted accurately and quickly.

Second, it would be interesting to evaluate the approach for optimizations that fit into the same framework as parallel interval analysis but are more complex. Examples are loop vectorization and software pipelining: non-nested loops can be treated in parallel, and the amount of work that must be performed in each parallel task is larger than the amount of work required to perform bit vector intersections in data flow analysis.

8.3.2. Global register allocation: future directions

We showed that our concept of using program structure for the parallelization of unstructured optimizations works well for global register allocation but there are few optimizations that fit into the specific framework of graph coloring. Our structured model for global register allocation serves as a basis for the improvement of spilling and splitting heuristics for global register allocation. This aspect of our model can only be evaluated when it is incorporated into the backend of a compiler.

Because of the importance of inter procedural optimizations, it would be interesting to extend the model to handle inter procedural register allocation. Like for loops and conditionals, this involves the development of boundary conditions for procedure calls. Further, the model can be adapted to handle register hierarchies and register conventions. One example of a register convention is the use of specific registers for parameters.

In general we believe that for well structured programs, compiler optimizations that are reduced to NP complete problems can be improved when knowledge about program structure is used when that NP complete problem is solved. For global register allocation the thesis demonstrated that this is possible.

References

[Aho 86] Aho, A.V., Sethi, R., Ullman, J.D.
Compilers: Principles, Techniques, and Tools.
Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[Bernstein et al 89]
Bernstein, D., Goldin, D.Q., Golumbic, M.C., Krawczyk, H., Mansour, Y., Nahshon, I., Pinter, R.
Spill Code Minimization Techniques for Optimizing Compilers.
In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 258-263. ACM SIGPLAN, June, 1989.

[Boehm 87] Boehm, H. J., and Zwaenepoel, W.
Parallel Attribute Grammar Evaluation.
In *7th Intl. Conf. on Distributed Computing Systems.* IEEE, Berlin, September, 1987.
Earlier published as Rice Tech. Report 86-39.

[Briggs et al 89] Briggs, P., Cooper, K.D., Kennedy, K., Torczon, L.
Coloring Heuristics for Register Allocation.
In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 275-284. ACM SIGPLAN, June, 1989.

[BubZwaen 92] Bubenik, R., Zwaenepoel, W.
Optimistic Make.
IEEE Transactions on Computers 41(2):207 - 217, February, 1992.

[CallahanKoblenz 91]
Callahan, D., Koblenz, B.
Register Allocation via Hierarchical Graph Coloring.
In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 192-203. ACM SIGPLAN, June, 1991.

[Chaitin 81] Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.
Register Allocation by Coloring.
Research Report 8395, IBM Watson Research Center, 1981.

[ChowHen 90] Chow, F.C., Hennessy, J.L.
The Priority-Based Coloring Approach to Register Allocation.
ACM Transactions on Programming Languages and Systems 12(4):501-536, October, 1990.

[Cooper 88] Cooper, E.E., Draves, R.P.
C Threads.
Technical Report, Carnegie Mellon University, June , 1988.

[Fischer 75] Fischer, C. N.
On Parsing Context Free Languages in Parallel Environments.
PhD thesis, Cornell University, 1975.

[Fishburn 85] Fishburn, P.C.
Interval Graphs and Interval Orders.
Discrete Mathematics (55):135-149, 1985.

[Frankel 83] Frankel, J. L.
The Architecture of Closely-Coupled Distributed Computers and their Language Processors.
PhD thesis, Harvard University, 1983.

[FreuRut 91] Freudenberger, S.M., Ruttenberg, J.C.
Phase Ordering of Register Allocation and Instruction Scheduling.
In *Code Generation - Concepts, Tools, Techniques*. Springer, May, 1991.
to appear.

[Garey, M.R. and Johnson, D.S. 79]
Garey, M.R. and Johnson, D.S.
Computers and Intractability: A Guide to the Theory of NP-Completeness.
W.H. Freeman, San Francisco, 1979.

[Gavril 72] Gavril, F.
Algorithms for Minimum Coloring, Maximum Clique, Minimum Covering by Cliques, and Maximum Independent Set of a Chordal Graph.
SIAM J. Computing 1(2):180-187, 1972.

[Gol 85] Golumbic, M.C.
Interval Graphs and Related Topics.
Discrete Mathematics (55):113-243, 1985.

[GrossSteenkiste 90]
Gross, T., Steenkiste, P.
Structured Dataflow Analysis for Arrays and its Use in an Optimizing Compiler.
Software-Practice and Experience 20(2), February, 1990.

[GrossZobel 89] Gross, T.R., Zobel, A., Zolg, M. .
Parallel Compilation for a Parallel Machine.
In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 91-100. ACM SIGPLAN, June, 1989.

[GupSofSte 89] Gupta, R., Soffa, M.L., Steele, T.
Register Allocation Via Clique Separators.
In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pages 264-274. ACM SIGPLAN, June, 1989.

[GuptaPollockSoffa 90]
Gupta, R., Pollock, L., Soffa, M.L.
Parallelizing Data Flow Analysis.
In *Proceedings of the First Workshop on Parallel Compilation*. May, 1990.

[HilLar 86] Larus, J.R., Hilfinger, P.N.
Register Allocation in the SPUR Lisp Compiler.
In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 255-263. ACM SIGPLAN, June, 1986.

[Kennedy 81] Kennedy, K.
A Survey of Data Flow Analysis Techniques.
In Muchnick, S. S. and Jones, N. D. (editors), *Program Flow Analysis*, chapter 1, pages 1-54. Prentice-Hall, New Jersey, 1981.

[Klein 90] Klein, E.
Attribute Evaluation in Parallel.
In *Proceedings of the First Workshop on Parallel Compilation*. May, 1990.

[KuckKuhnEtAl 81]
Kuck, D. J., Kuhn R. H., Padua, D. A., Leasure, B., and Wolfe, M. .
Dependence Graphs and Compiler Optimizations.
In *Conference Record of the 8th Annual ACM Symposium on Programming Languages*, pages 207-218. ACM, Williamsburg, January, 1981.

[Kung 88] Kung, H. T.
Warp Experience: We Can Map Computations onto a Parallel Computer Efficiently.
In *Conference Proceedings of 1988 International Conference on Supercomputing*, pages 668-675. ACM, St. Malo, France, July, 1988.

[Lam 88] Lam, M.
Software Pipelining: An Effective Scheduling Technique for VLIW Machines.
In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318-328. June, 1988.

[LeeMarloweRyder 91]
Lee, Y.-F., Marlowe, T.J., Ryder, B.G.
Experiences with a Parallel Algorithm for Data Flow Analysis.
The Journal of Supercomputing 5(2):163-188, Oct, 1991.

[McMahon 86] McMahon, F. H.
The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range.
Technical Report UCRL-53745, University of California, Lawrence Livermore National Laboratory, December, 1986.

[MorelRenvoise 81]
Morel, E. and Renvoise, C.
Interprocedural Elimination of Partial Redundancies.
In Muchnick, S. S. and Jones, N. D. (editors), *Program Flow Analysis*, chapter 6, pages 160-188. Prentice-Hall, New Jersey, 1981.

[Multimax 88] Encore Computer Corporation.
Multimax Technical Summary.
Technical Report, Encore Computer Corporation, , 1988.

[Pat/Hen 90] J. L. Hennessy and D. A. Patterson.
Computer Architecture A Quantitative Approach.
Morgan Kaufman, 1990.

[Pomerleau, et al. 88]
Pomerleau, D. A., Gusciora, G. L., Touretzky, D. S. and Kung, H. T.
Neural Network Simulation at Warp Speed: How We Got 17 Million Connections Per Second.
In *Submitted to the IEEE Second International Conf. on Neural Networks.* April, 1988.

[Press et al 88] Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T.
Numerical Recipes in C - The Art of Scientific Computing.
Cambridge University Press, 1988.

[Seshadri et al 88] Seshadri, V., Wortman, D.B., Junkin, M. D., Weber, S., Yu, C.P., and Small, I. .
Semantic Analysis in a Concurrent Compiler.
In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 233-239. ACM SIGPLAN, June, 1988.

[Sites 78a] Sites, R.
Instruction Ordering for the Cray-1 Computer.
Technical Report 78-CS-023, University of California, San Diego, July, 1978.

[Sites 78b] Sites, Richard L.
CRAY-1 Register Allocation for Optimized Pascal.
Technical Report, University of California at San Diego, October, 1978.

[Smith 87] Harry F. Smith.
Data Structures Form and Function.
Harcourt Brace Jovanovich, 1987.

[Stallman 88] Stallman, R.
Internals of GCC
Cambridge, Mass, 1988.

[Tarjan 85] Tarjan, R.E.
Decomposition by Clique Separators.
Discrete Math. 55:221-231, 1985.

[Vandevoorde 88] Vandevoorde, M. T.
Parallel Compilation on a Tightly Coupled Multiprocessor.
Master's thesis, MIT, March, 1988.

[Wall 86] Wall, D. W.
Global Register Allocation at Link Time.
In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 264-275. ACM SIGPLAN, June, 1986.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admissions and employment on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders. In addition, Carnegie Mellon University does not discriminate in admissions and employment on the basis of religion, creed, ancestry, belief, age, veteran status or sexual orientation in violation of any federal, state, or local laws or executive orders. Inquiries concerning application of this policy should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.