2

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

ON INCREASING THE EFFECTIVE BLOCKING FACTOR OF A MATRIX FOR A GIVEN CACHE ORGANIZATION

by

Atilla N. Demirhan

September 1992

Thesis Advisor: Amr Zaky

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School — 6b. OFFICE SYMBOL CS	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION — 8b. OFFICE SYMBOL	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Including Security Classification)
On Increasing the Effective Blocking Factor of a Matrix for a Given Cache Organization

12 PERSONAL AUTHOR(S)
Atilla N. DEMIRHAN

13 TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. Page Count
Master's thesis	FROM TO	1992, SEPTEMBER	72

16. SUPPLEMENTAL NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Self-interference Misses, Blocking, Tiling, Cache, Array Size, Performance

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Blocking (Tiling) techniques of iteration spaces to increase data reuse in the cache were review[illegible]ults consistent with those previously published were experimentally obtained. The relation between the size [illegible] the declared matrix and the cache was studied. Based on this relation, two algorithms were presented. Both algorithms attempt to increase the critical blocking factor with no self-interference (B_C) by changing the declared matrix size. Furthermore, the execution time of the second algorithm is independent of the matrix size. Experiments based on these algorithms were performed which showed a consistent superior performance (in terms of Mflops) relative to the performance obtained using previously published algorithms for deriving B_C.

20 DISTRIBUTION/AVAILABILTIY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC	1a. REPORT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Amr Zaky	22b. TELEPHONE (Include Area Code) (408)646-2174 — 22c. OFFICE SYMBOL CS/37

DD Form 1473, JUN 86 Previous editions are obsolete. S/N 0102-LF-014-6603

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

ON INCREASING THE EFFECTIVE BLOCKING FACTOR OF A MATRIX FOR A GIVEN CACHE ORGANIZATION

by

Atilla Demirhan
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1986

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1992

Author: [illegible]
Atilla Demirhan

Approved By: [illegible]
Amr Zaky, Thesis Advisor

[illegible]
Man-Tak Shing, Second Reader

[illegible]
Robert B. McGhee, Chairman,
Department of Computer Science

ABSTRACT

Blocking (Tiling) techniques of iteration spaces to increase data reuse in the cache were reviewed. Results consistent with those previously published were experimentally obtained. The relation between the sizes of the declared matrix and the cache was studied. Based on this relation, two algorithms were presented. Both algorithms attempt to increase the critical blocking factor with no self-interference (B_C) by changing the declared matrix size. Furthermore, the execution time of the second algorithm is independent of the matrix size. Experiments based on these algorithms were performed which showed a consistent superior performance (in terms of Mflops) relative to the performance obtained using previously published algorithms for deriving B_C.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. PERFORMANCE IMPROVEMENT IN A COMPUTER

1. The Memory Bottleneck

An essential component of every computer is its memory. Without memory there could be no computers as we now know them. There is an important axiom of hardware design: **smaller is faster** [Ref. 18]. Smaller pieces of hardware will generally be faster than larger pieces. In high-speed machines, signal propagation is a major cause of delay, and larger memories have more signal delay and require more levels to decode addresses. A basic attribute used to measure the effectiveness of a memory configuration is the memory bandwidth. This is the number of words that can be accessed per second or the rate of information transfer. Increasing the computational power without a corresponding increase in the memory bandwidth of data to and from memory can create a serious bottleneck [Ref. 18]. In other words, *increasing memory bandwidth* and *decreasing the latency(execution time) of memory access* are both crucial to system performance. Since these two measures are so much important, we should improve them to obtain a better performance.

2. Cache Memory

The analysis of a large number of computer programs has shown that during program execution, memory references tend to occur in very localized patterns. Consider, for example, a block of contiguous memory locations. If this block consists of straight line code, then execution will proceed sequentially through the block. If the block represents a data array, then it is likely that the block will often be used, not necessarily in a sequential order, but at least as a whole. This characteristic of programs is referred to as the **locality of reference principle.** When a program is stored in main memory, the access time for fetching a memory word is a function of the capacity(size) of the memory and not a function of the size of a local block in the memory [Ref. 19]. Hence, to get the benefit from the locality of reference principle, high-speed buffer(s) can be inserted between the processor(s) and main memory to capture those active portions (blocks) of the contents of main memory currently in use. Then, rather than fetching the word at the next location from the main memory, it could be fetched much more quickly from this special high-speed buffer called **cache memory** and implemented in most computers as shown in Figure 1. The major reasons for having a memory hierarchy are to get a better performance and to reduce execution time, not misses [Ref. 18]. If cache misses can be reduced by some means, then the overall performance of processors will improve.

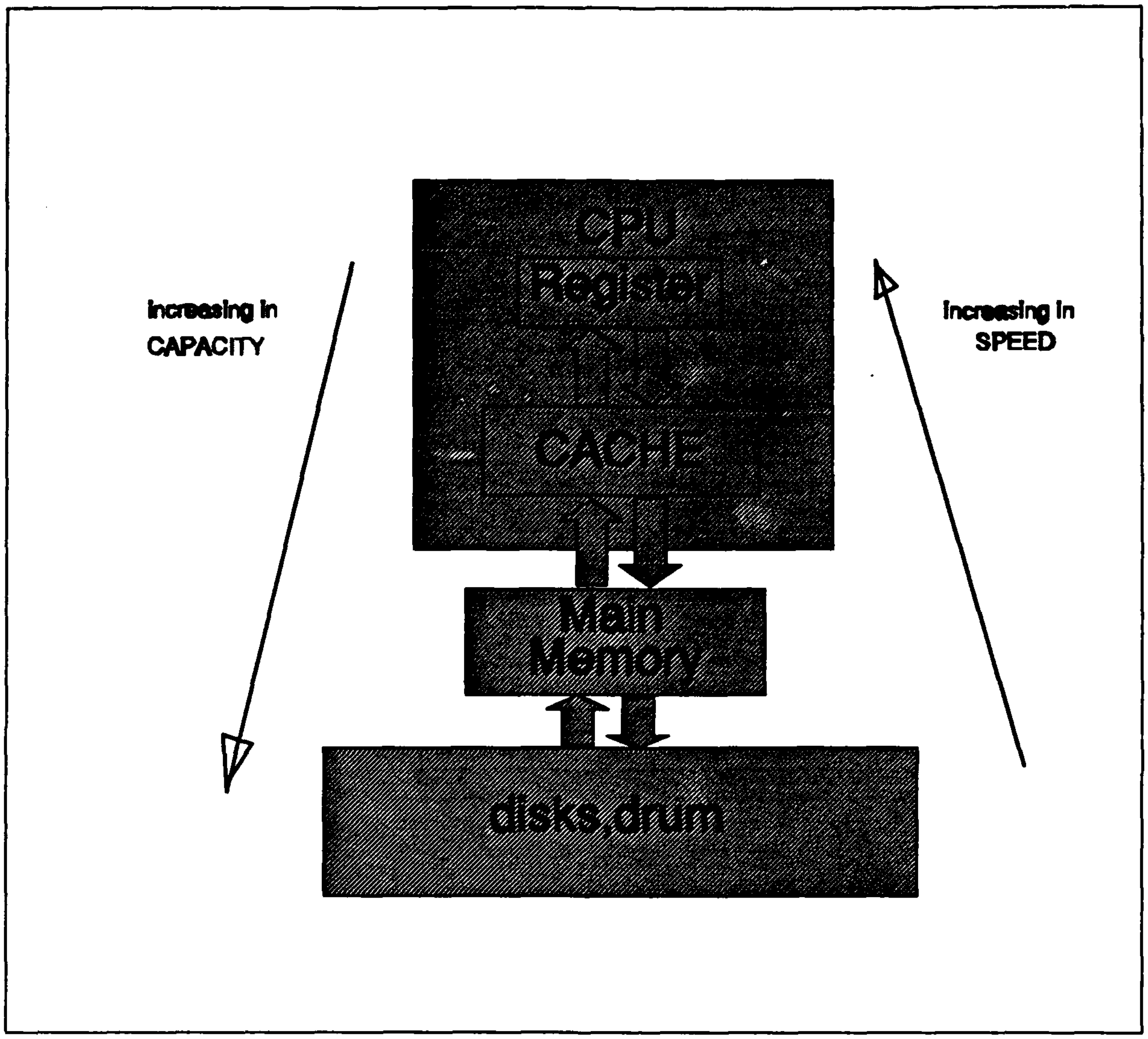

Figure 1

Memory Hierarchy

To fully realize the potential of the processors, the faster levels of the memory hierarchy such as *cache memory* must be efficiently utilized. But, a distinct characteristic of numerical applications is that they tend to operate on large data sets where as a cache may only be able to hold small fraction of a matrix. Thus even if the data are reused, they may have been displaced from the cache by the time they are reused, causing a high miss ratio in the cache. For this reason, **blocking techniques** are utilized for the goal of reducing memory traffic and exploiting this data reuse. Blocking techniques restructure the code to move references to the same memory location closer together, hence improving cache performance. Blocking can be applied at different levels of the memory hierarchy, including physical memory, caches, and registers [Ref. 9]. In the experiments, blocking has been applied both at data cache and register levels.

B. IMPROVING DATA LOCALITY

1. Types of Misses in Cache

Cache misses occur in one of three forms:

* *Compulsory misses:* The first access to a block is not in the cache, so the block must be brought into the cache. These are called *cold start misses* or *reference misses.*

* *Capacity misses:* If the cache cannot contain all the blocks needed during a program execution of a program, capacity misses will occur due to blocks being discarded and later retrieved.

* *Conflict misses:* If the block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. These are also called *collision misses* [Ref. 19].

Conceptually, conflict misses could be the easiest to avoid: Fully associative placement avoids all conflict misses by mapping an address in the main memory to any cache block. But, this associativity is expensive in hardware and may also slow down the access time, leading to a lower overall performance [Ref. 19]. Therefore, fully associative caches are not built. Capacity misses can be reduced by using a large cache. Larger cache lines reduce the number of compulsory misses, but this may lead to an increase in conflict misses.

2. Transformation (Restructuring)

If a program can be transformed into a version that makes better use of the cache, then the number of requests to memory can be reduced, thus improving the execution time. Program transformations that move consecutive uses of a memory location closer together can increase the number of times a value is used before it is replaced. A cache miss removed by code transformation will require less traffic since the number of times a value is loaded into the cache is decreased [Ref. 7].

The basic theory of this transformation (restructuring) process is based on *data dependence analysis*. Data dependence information is needed to test whether restructuring transformations are legal (the program produces the same answer after restructuring as it did before) [Ref. 13]. Data dependence analysis will be explained in detail in Chapter II. Briefly, it is a tool which is applied to partition a serial program into blocks of code that contain well-defined dependences. The purpose of dependence analysis is to prove the presence or absence of data dependence between the blocks [Ref. 16]. These blocks may be converted into independent tasks, scheduled onto one or more parallel processors, and run as a parallel program. The problem gets complex and difficult when the blocks are control structures such as loops, branches, and procedures.

Loops are difficult to analyze, and yet they are the best candidate for optimizations obtained by **transformation techniques** such as *vectorization, fusion, coalescing, distribution, interchange, node splitting, shrinking, unfolding, skewing, and* ***blocking transformation techniques*** [Ref. 16].

3. Performance Experiments with Blocked Codes

In this thesis, we experiment with techniques to improve data cache performance with blocked codes. We apply these techniques on two machines: *DECstation 3100* and *Solbourne S4000 SPARC station.* Together with blocking, we investigate the effect of further optimizations such as *unfolding (unrolling)* on the data cache performance.

Blocking techniques are used to increase the life time of the piece of data in the cache. Therefore, this allows reused data to still be in the cache and reduces the memory accesses. Blocking techniques improve cache performance by restructuring the code to move references to the same memory location closer together, hence eliminating cache misses. To illustrate the benefits of blocking, we handle matrix multiplication code, showing how it is blocked and how the reuse of data in that piece of code can be exploited. A basic matrix multiplication computes an inner product of a row and a column of matrices B and C for each element of the result matrix A.

```
for (I = 1; I<N; ++I) {
    for (J = 1; J<N; ++J) {
        for (K = 1; K<N; ++K) {
            C(I,J) += A(I,K)*B(K,J);
        }
    }
}
```

Figure 2

Main loops of Matrix Multiplication

We assume that we have a direct-mapped data cache and the arrays are stored rowwise in cache lines. If we substitute some loop bounds for the code above, then we can show the potential in it for the reuse easily. In Example 1 below, matrix multiplication code is utilized and N has the value 2.

Example 1:

```
    C(1, 1) += A(1, 1) * B(1, 1)    [I = 1, J = 1 - 2, K = 1]
    C(1, 2) += A(1, 1) * B(1, 2)
-------------------------------------------------------------
    C(1, 1) += A(1, 2) * B(2, 1)    [I = 1, J = 1 - 2, K = 2]
    C(1, 2) += A(1, 2) * B(2, 2)
-------------------------------------------------------------
    C(2, 1) += A(2, 1) * B(1, 1)    [I = 2, J = 1 - 2, K = 1]
    C(2, 2) += A(2, 1) * B(1, 2)
-------------------------------------------------------------
    C(2, 1) += A(2, 2) * B(2, 1)    [I = 2, J = 1 - 2, K = 2]
    C(2, 2) += A(2, 2) * B(2, 2)
-------------------------------------------------------------
```

As seen above, there are array variables which are repeatedly referenced and retrieved from the memory. For example, both A(1, 1) is referenced when K = 1 for different iterations of J loop. A(2, 2) present the same behavior when K = 2. All these repeating references cause unwanted cache traffic (if the cache is not large enough to hold the whole A array). Here we assume that we have a data cache size of $2^2 = 4$ words and the reused array elements in the first four lines of Example 1 are placed in this cache as shown in Figure 3.

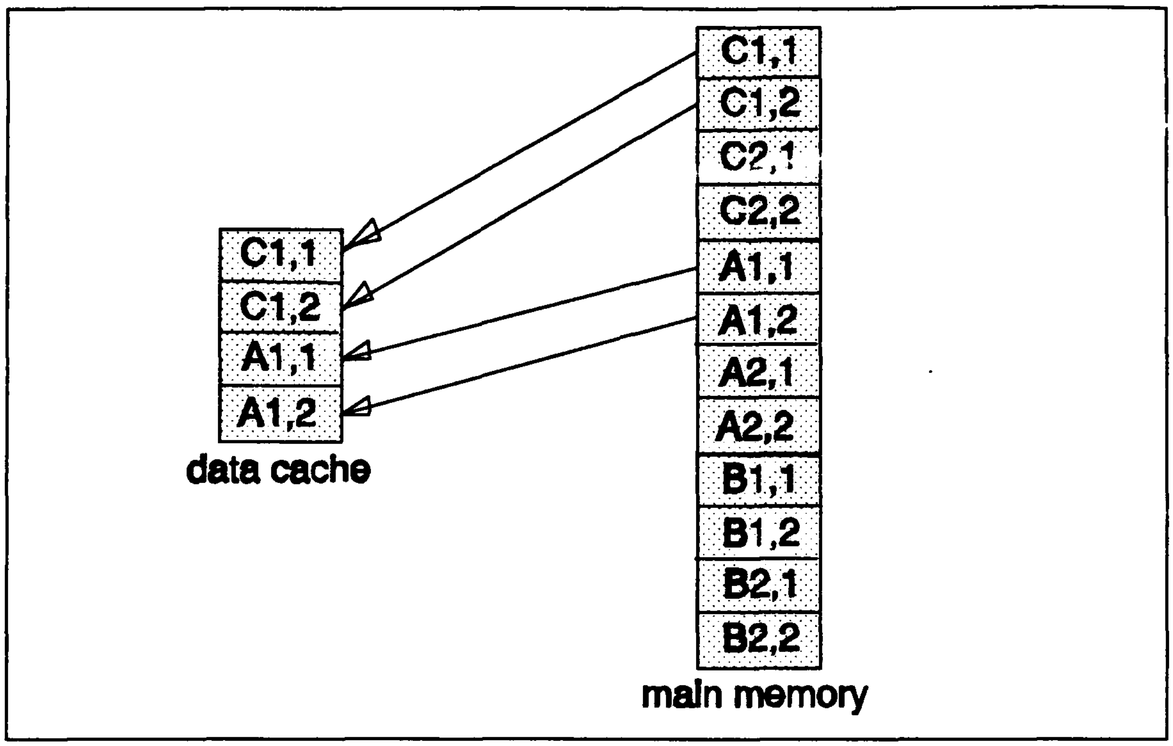

Figure 3

Data Cache and Main Memory

To provide this type of placement in the ca[illegible] we can use subblocks of the arrays as follows:

$$C^{1,1} = A^{1,1} * B^{1,1} + A^{1,2} * B^{2,1}$$

$$C^{1,2} = A^{1,1} * B^{1,2} + A^{1,2} * B^{2,2}$$

$$C^{2,1} = A^{2,1} * B^{1,1} + A^{2,2} * B^{2,1}$$

$$C^{2,2} = A^{2,1} * B^{1,2} + A^{2,2} * B^{2,2}$$

The following figure shows these subblocks explicitly.

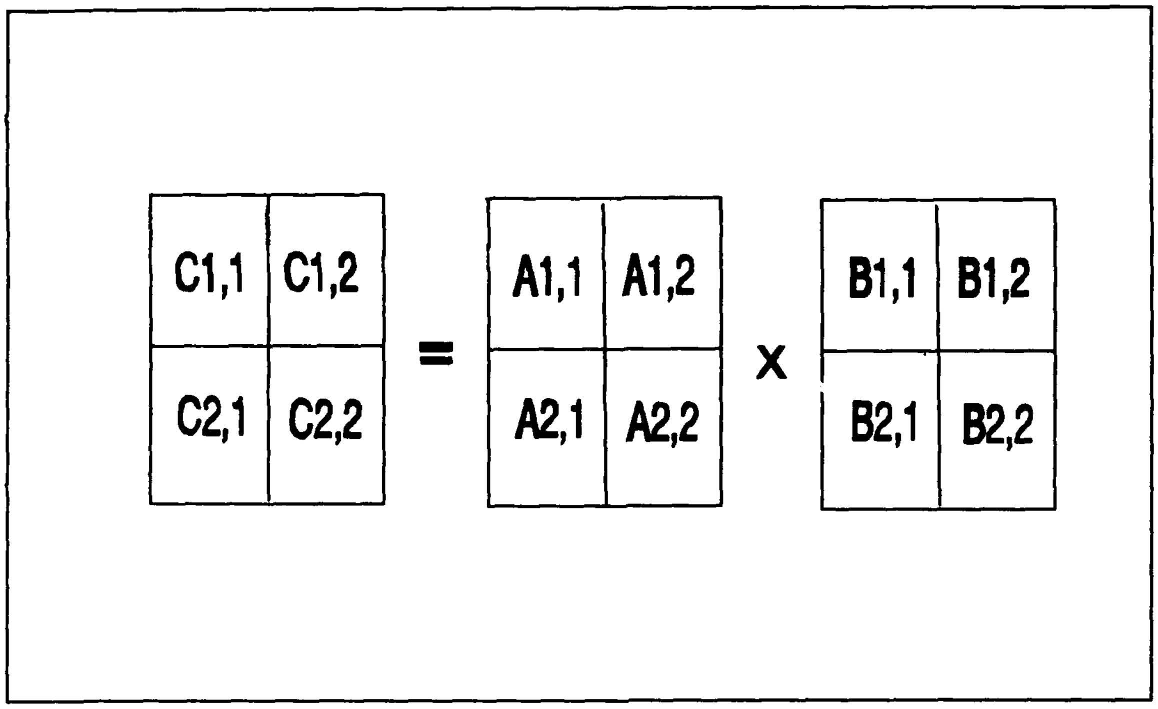

Figure 4

Subblocks in a matrix multiplication code when matrix size is 2

This subblocking exhibits the advantage that the blocks can be sized to fit into the fastest level of the memory hierarchy such as cache memory, and by this way the data in each submatrix is used many times during each matrix multiplication. These benefits of subblocking can be enhanced via program transformations which form the basis of *blocking techniques*. The main loops of matrix multiplication code in Figure 2 are blocked and shown in Figure 5.

```
for (KK=0; KK<N; KK=KK+B) {
    for (JJ=0; JJ<N; JJ=JJ+B) {
        for (I=0; I<N; ++I) {
            for (K=KK; K<min(N,KK+B); ++K) {
                C[I][J] = 0;
                temp = A[I][K];
                M  = min(N,JJ+B);
                for (J=JJ; J<M; ++J) {
                    C[I][J] = C[I][J] + temp*B[K][J];
                }
            }
        }
    }
}
```

Figure 5

Main Loops of Blocked Matrix Multiplication

4. Previous Work on Blocking

It has long been known that program restructuring can dramatically reduce the load on a memory hierarchy subsystem ([1], [2], [3]). Previous research on blocking focused on how to block an algorithm manually and automatically ([5], [6], [7], [8]). Irigoin and Triolet [Ref. 4] describe a procedure to partition the iteration space of a tightly-loop into supernodes, where each supernode covers a set of iterations that will be scheduled as an atomic task on a processor. That procedure works from a new data dependence abstraction, called the dependence cone. The dependence cone is used to find legal partitions and to find dependence constraints between supernodes.

Lam, Rothberg, and Wolf [Ref. 9] show that the degree of cache *interference* is highly sensitive to the stride of data accesses and the size of the blocks, and can cause wide variations in machine performance for different matrix sizes. They presented cache performance data (obtained via simulation) for blocked programs and evaluate several optimizations such as using a fixed blocking factor and copying non-contiguous data to be reused into a contig-uous area. They found that trying to use the entire cache, or even a fixed fraction of the cache is not so useful and propose an algorithm to tailor the block size according to matrix size to improve the average miss rate.

Hong and Kung [Ref. 10] claim that the optimal blocking factor is roughly SQRT(C) for matrix multiplication on a machine with a local memory of C words.

In an algorithm implemented in Stanford University Intermediate Format Compiler [Ref. 8], the locality of a loop nest by transforming the code via interchange, reversal, skewing, and blocking is improved.

C. ORGANIZATION OF THE THESIS

In Chapter II, we focus on existing blocking techniques, by explaining their advantages and related terminology. We also present data obtained from the experiments with blocked codes to observe the effectiveness of these blocking techniques. In Chapter III, we propose an enhanced blocking technique which improves the performance of workstations in some scientific applications. Conclusions and recommendations for further research are offered in Chapter IV.

II. EXPERIMENTING WITH BLOCKING

A. BLOCKING TECHNIQUES

1. Program Restructuring

Blocking techniques improve cache performance by restructuring the code, hence eliminating cache misses. Advanced compilers or supercompilers are capable of many program restructuring transformations such as vectorization, strip mining and loop interchanging [Ref. 13]. These transformations transform a program into a version that makes better use of the cache, thus the number of requests to memory is reduced and the execution time is improved. But super-compilers cannot transform every program efficiently; the quality of the results will depend on the structure of the algorithm and the architecture of the target machine. Data dependences [Ref. 21] imply precedence constraints among computations which have to be satisfied for a correct execution. So, when determining the loops for restructuring, the existence of these dependences must be considered. In the next sections, concepts such as *data dependence, data dependence graph* and *iteration space* will be described.

a. *Data Dependence*

Two types of dependence occur in computer programs. *Control Dependence* is a consequence of the flow of control in a program. Execution of a statement in one path under an **if** test is dependent on the **if** test taking the path. Thus, the statement under control of the **if** is *control dependent* upon the **if** test. *Data Dependence* is a consequence of the flow of data in a program. The value of an expression is dependent upon the values of the variables used in the expression. Therefore, a statement which uses a variable in an expression is *data dependent* upon the statement which computes the value of the variable.

In programming languages, such as C, Fortran, and Pascal, three kinds of data dependence may occur: *flow-dependence(or true dependence)*, *anti-dependence*, and *output-dependence*. The first dependence relation occurs when a value computed(stored) in a statement S_v is used(fetched) in some statement S_w; we say that S_w is *data flow-dependent* on S_v and write this as S_v & S_w. This type of data dependence relation shows how the data flows between the statements of the program. The second kind of data dependence occurs when an item is used in statement S_v before that item is reassigned in some statement S_w; we say that S_w is *data anti-dependent* on S_v before that item is reassigned in some statement S_w and write this as S_v &$^-$ S_w. The last kind of data dependence occurs when an item is assigned in statement S_v before that item is reassigned in some statement S_w; we say that S_w is *data output-dependent* on S_v and write this as S_v &$^\circ$ S_w [Ref. 21]. We illustrate these dependencies in Example 2.

Example 2.

```
S1 : A = B + D
S2 : C = A * 3
S3 : A = A + C
S4 : E = A / 2
```

The dependence relations for this code are:

S2,S3,S4 are data flow-dependent on S1,S2,S3 respectively;

S3 is also data flow-dependent on S1;

S3 is data anti-dependent on S2;

S3 is data output-dependent on S1.

b. Dependence Graph

A dependence relation is a precedence relation which requires execution of one statement before another. A parallelizing (or optimizing) compiler can build *a dependence graph* by using these dependences. In a dependence graph, nodes represent statements in the program and directed arcs represent dependence relations [Ref. 21]. The real advantage of dependence graphs is their ignoring the arbitrary ordering of statements and concentrating on the dependence precedence. The dependence graph for Example 2 is depicted in Figure 6:

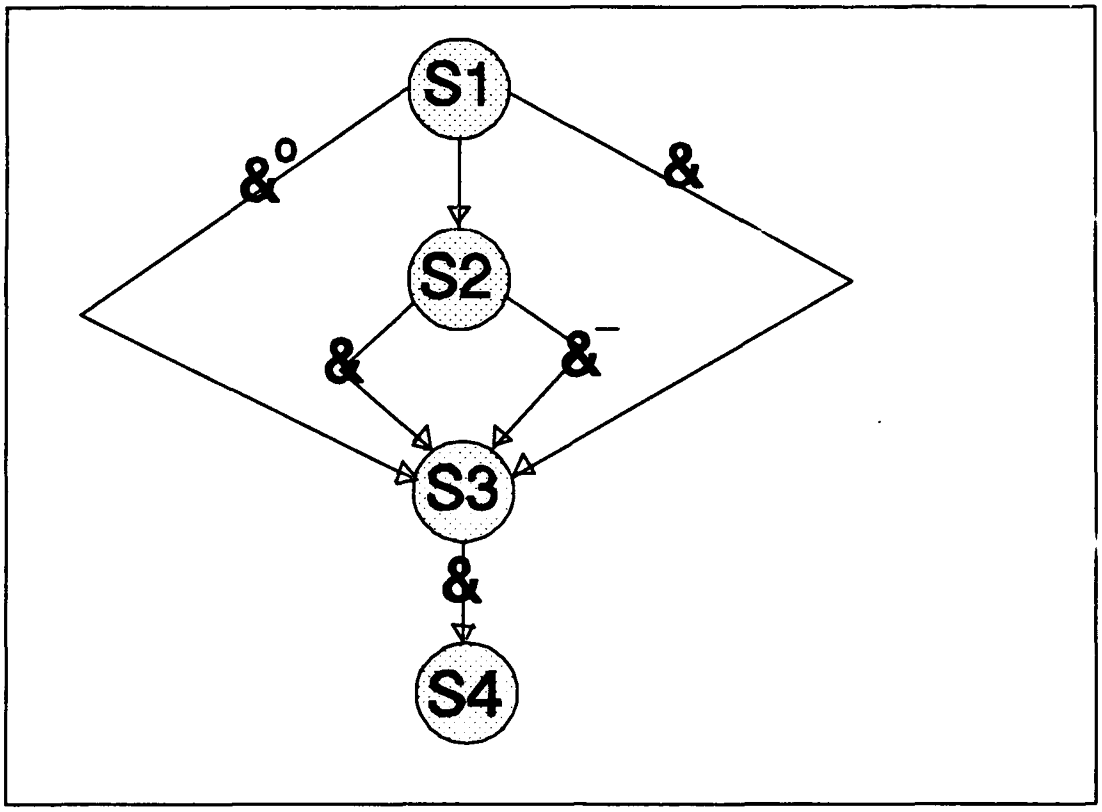

Figure 6

Dependence Graph for the code Example 2

c. Iteration Space

A loop, when there is considerable potential for code optimization, can be said to describe an iteration space. A single **for** loop describes a one-dimensional iteration space, one axis of a Cartesian coordinate system. Each iteration of the **for** loop corresponds to a point along this axis. The **for** loop will visit the points along this axis in a specific order, as defined by the **for** statement. If we have two nested **for** loops, then they describe a two-dimensional iteration space.

```
        for (I1 = 1; I1<5; ++I1) {
            for (I2 = 1; I2<4; ++I2) {
S1 :            A(I1, I2)    = B(I1, I2) + C(I1, I2)
S2 :            B(I1, I2+1) = A(I1, I2) + B(I1, I2)
            }
        }
```

Figure 7

Two-nested serial loops

The loops above define a two-dimensional 5X4 iteration space illustrated in Figure 8.

		I1			
		1	2	3	4
	1	x	x	x	x
	2	x	x	x	x
I2	3	x	x	x	x
	4	x	x	x	x
	5	x	x	x	x

Figure 8

Iteration space for the code in Figure 7

The shape of the iteration space does not need to be rectangular. The iteration space for the loops below is triangular, so it is called as a triangular loop. Other iteration space shapes can be defined by nested loop.

```
for (i = 1; i<5; ++i) {
   for (j = i; j<10; ++j) {
       A(i,j) = B(i,j) + C(i,j) * D(i,j)
   }
}
```

Figure 9

Triangular Loop

In this research, a loop transformation technique **blocking (iteration space tiling)** is implemented on two different machines. The code used in the experiments is optimized by utilizing blocking optimizations to get better speedups. We also utilized another optimization technique called *loop unfolding* on the blocked code but it did not help the performance of the data cache. Blocking utilizes two other transformation techniques, **loop interchanging** and **strip mining.**

d. Loop Interchanging

One of the most important restructuring transformations is *loop interchanging.* The simplest example of loop interchanging is interchanging two-nested loop, such as the loop below:

```
for (I = 1; I<N; ++I) {
   for (J = 1; J<N; ++J) {
      A(I,J+1) = A(I,J) * B(I,K) + C(K,J)
   }
}
```

Figure 10

Two-Nested Loops

The nested loop above is a first order linear recurrence; most vector computers have no corresponding instruction, so the loop would be executed serially, or with some fast recurrence algorithm. By interchanging the loops:

```
for (J = 1; J<N; ++J) {
   for (I = 1; I<N; ++I) {
      A(I,J+1) = A(I,J) * B(I,K) + C(K,J)
   }
}
```

Figure 11

Two-Nested Loops After the Interchange

The inner loop is vectorizable. Not all loop interchanges are legal. For instance, the loops below can not be interchanged:

```
for (I = 2; I<N; ++I) {
   for (J = 1; J<N-1; ++J) {
      A(I,J) = A(I-1,J+1) + B(I,K)
   }
}
```

The original loop uses newly computed values of the array A on the right hand side; if the loops are interchanged, the transformed loop will use only old values of A on the right hand side. The transformed loop will compute different results and the transformation is therefore illegal.

e. Strip Mining

Vectorizing compilers often divide a single loop into a pair of loops, where the maximum trip count of the inner loop is equal to the maximum vector length of the machine. The loop in Example 3.a. can be converted into a pair of the loops in Example 3.b by a vectorizing compiler. This process is called *strip mining.* The original loop is divided into strips of some maximum size, the strip size, in Example 3.b., the inner loop (or element loop) has a strip size of *B*, which is the length of the vector register in the compiler. The outer loop (IS loop, for strip loop) steps between the strips;

Example 3.a.

```
          for (I = 1; I<N; ++I) {
$S_1$ :        A(I) = A(I) + B(I)
$S_2$ :        C(I) = A(I-1) * 2
          }
```

Example 3.b.

```
            for (IS = 1; I<N; IS = IS + B) {
                for (I = IS; I<min(N, IS + B); ++I) {
    S1 :            A(I) = A(I) + B(I)
    S2 :            C(I) = A(I-1) * 2
                }
            }
```

2. Blocking (Iteration Space Tiling)

a. *Blocking Algorithms*

Blocking algorithms have been developed for the goal of reducing memory traffic. Its advantage is that the blocks can be sized to fit into the fastest level of the memory hierarchy, and that during each computation, the data in each block is used many times. Blocking tries to duplicate the benefits of block algorithms via program transformations.

b. *Blocking (Tiling)*

We define *Blocking* as dividing the iteration space into *blocks (tiles)* of some size and shape (typically squares or cubes), and traversing between the blocks to cover the whole iteration space. Optimal blocking for a memory hierarchy will find blocks such that all the data for each block will fit into the highest level of the memory hierarchy [Ref. 13]. This will reduce the number of intervening iterations and data fetched between data reuses. The reused data will still be in the cache, and hence memory accesses will be reduced. In other words, the traversal between the blocks will follow an order that will reduce the amount of data that needs to be moved when going to the next tile. To be most effective at blocking (tiling), block size must be tuned to fit into cache memory to allow the minimum number of misses to occur and generate the least traffic between the memory levels.

Blocking, as a combination of strip mining and loop interchanging, can improve codes. The goal is to strip the innermost loop into pieces such that the new innermost loop fits entirely into cache memory. The newly created middle loop is then moved to the outer loop. Figure 12 below illustrates how blocking is applied to loops whenever it is legal.

```
    for (I = 1; I<N; ++I) {
        for (J = 1; J<N; ++J) {
            loop body
        }
    }
becomes
      for (JJ = 1; JJ<N; JJ = JJ + B) {
          for (I = 1; I<N; ++I) {
               for (J = JJ; JJ<B; ++J) {
                    loop body  }  }  }
```

Figure 12

Blocking. B is the strip size

A further optimized version of the code in Figure 12 is illustrated in Figure 13. The inner two loops iterate in square shaped blocks (BxB) while the outer two loops step between the blocks.

```
for (II = 1; II<N; II = II + B) {
     for (JJ = 1; JJ<N; JJ = JJ + B) {
          for (I = II; I<min(N, II + B); ++I) {
               for (J = JJ; J<min(N, JJ + B); ++J) {
                    loop body
               }
          }
     }
}
```

Figure 13

Further Optimized version of the code in Figure 12

The iteration space for the above code is shown in Figure 15.

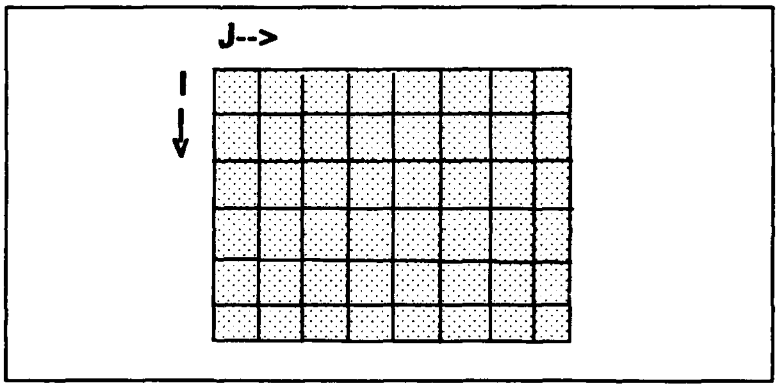

Figure 14

Iteration Space for the code in Figure 13

B. PERFORMANCE EXPERIMENTS WITH BLOCKED MATRIX MULTIPLICATION CODE

The experiments are performed on two different machines located in The Naval PostGraduate School *Matrix multiplication* is utilized as the main source code written in C language. Matrix multiplication is a building block in many linear algebraic algorithms. It is also an interesting case study because locality is carried in three different loops by three different variables. The entire computation in a matrix multiplication involves $2N^3$ arithmetic operations (counting additions and multiplications separately), but produces and consumes only $3N^2$ data values. As a whole, the computation exhibits "admirable reuse of data" [Ref. 23]. In general, however, an entire matrix will not fit in a small data cache memory. So, the code is restructured such that the necessary reuse of data is achieved in cache.

1. Experimental Setup

In our experiments, three different matrix sizes (293, 295, 300)[1] are utilized and the performance of blocked matrix multiplication is observed by experimenting with square shaped blocks having sizes of multiples of 8. To ensure the accuracy of the timing results obtained, each experiment has been run 5 times by using the library function *system()* which provides access to operating system commands.

[1]These values were chosen so that we compare our results to those in [Ref. 9].

The "MFLOPS" rating of the machine was utilized as the performance measure for the blocked portion of the code, i.e., the loops. We had to find the time of that portion. The command *"time"* provided in C compiler is not appropriate to measure for that type of computation. Because, it is an executable program available when using the shell and it measures the time of an executable program from the beginning to the end. So, in our experiments, another structure called *getrusage* is used. *"getrusage"* returns the user time utilized by the current process, or all its terminated child processes. By calling it twice, at the beginning and at the end of the nested loops, we measured the time we were looking for. The code used in the experiments is presented in Figure 15 below.

```
float Timing1, Timing2, Difference, Performance;
struct rusage *x, *y; /* declaration */
/* other declarations and lines of code will be here*/
x = (struct rusage *) malloc (sizeof(struct rusage));
y = (struct rusage *) malloc (sizeof(struct rusage));
if (getrusage(RUSAGE_SELF, x) == -1)
    perror("getrusage()");
/* Nested loops are put here */
if (getrusage(RUSAGE_SELF, y) == -1)
    perror("getrusage()");
Timing1 = (y->ru_utime.tv_usec - x->ru_utime.tv_usec)/1000;
Timing2 =  y->ru_utime.tv_usec - x->ru_utime.tv_usec);
          if (Timing1 < 0) {
            Timing2 = Timing2 - 1;
            Timing1 = 1000 + Timing1;
        }
Difference = Timing2 + Timing1 / 1000;
Performance = (2*N*N*N) / (1000000*Difference); /* MFLOPS */
```

Figure 15

The code to compute the performance of the cache

During the experiments, we needed to edit files quickly to change the values for matrix and block size. We intensely used the Unix editor, *sed,* to make changes to the files on the command line. Also, usage of *Tschell* (shell) together with *sed* allowed the task of making frequent changes to the source files by the processor less tedious.

2. Overview of The Targeted Machine Architectures

The two machines used in the experiments are *Solbourne S4000 SPARC Station* and *DECstation 3100. The Solbourne S4000 SPARC Station* has a 64-bit SPARC CPU, with a 2kB 2-way set-associative write-back physical data cache (DCACHE) and 6kB 3-way set-associative instruction cache (ICACHE). The processor performs a load/store instruction in one clock, achieving 25.5 MIPS. In SPARC microprocessor, all performance-critical element--integer CPU, floating point processor, memory management unit and cache--are integrated on a single chip. The 64-bit memory bus supports a data transfer rate of 60 Mbytes per second to accommodate the high performance requirements of the processor. The *DECstation 3100* has an 8kB double word direct-mapped cache and can hold all the words reused within an 88x88 block. For this reason, experiments are also done with block sizes over 88x88 in order to measure the real performance. Otherwise, the data cache can hold 88x88 (or smaller) blocks and the effects of self-interference misses may not be observed.

3. Blocking Experiments

The main loops of unblocked and blocked matrix multiplication are illustrated in Figure 16 and 17 respectively.

```
for (i=0; i<N; ++i) {
    for (j=0; j<N; ++j) {
        for (i=0; i<N; ++i) {
            c[i][j] = c[i][j] + temp*b[k][j];
        }
    }
}
```

Figure 16

Main Loops of Unblocked Matrix Multiplication

```
for (kk=0; kk<N; kk=kk+B) {
    for (jj=0; jj<N; jj=jj+B) {
        for (i=0; i<N; ++i) {
            for (k=kk; k<min(N,kk+B); ++k) {
                c[i][j] = 0;
```

```
temp = a[i][k];
M  = min(N,jj+B);
for (j=jj; j<M; ++j) {
    c[i][j] = c[i][j] + temp*b[k][j];
.....
```

Figure 17

Main Loops of Blocked Matrix Multiplication

First, the code in Figure 17 is blocked at both data cache and register levels. a[i][k] is register allocated and this relatively increases the performance of blocked code. Secondly, *min* function is handled with a macro. In the very first experiments, the *min* function was placed in the *for* statement (as in Figure 18).

```
.....
temp = a[i][k];
     for (j=jj; j<min(N,jj+B); ++j) {
          c[i][j] = c[i][j] + temp*b[k][j];
.....
```

Figure 18

The *for* statement with *min* function in it

Removing it as in Figure 19 improved the performance drastically.

```
.....
temp = a[i][k];
        M  = min(N,jj+B);
        for (j=jj; j<M; ++j) {
             c[i][j] = c[i][j] + temp*b[k][j];
.....
```

Figure 19

The *for* statement without *min* function in it

a. *Cache Performance Experiments with Blocked Codes on Solbourne S4000 SPARC station*

In Figure 20, the performance of blocked matrix multiplication is plotted on Solbourne S4000 SPARC station. The code is compiled without enabling the code optimization of *gcc* (the *gnu* compiler). The graph in Figure 20 plots the performance levels obtained for three slightly different matrix sizes across a range of blocking factors. Blocking effects are not observed when code optimization of the compiler is not enabled. There is a negligible change(<6%) between performances of blocked and non-blocked code when no optimization switch is used. The reason is that the number of instructions in the non-optimized code was much bigger than that of the optimized one and the percentage of memory access instructions to total memory instructions was very small. In other words, the code was running inefficiently. In contrast, optimized code had a smaller number of memory instructions

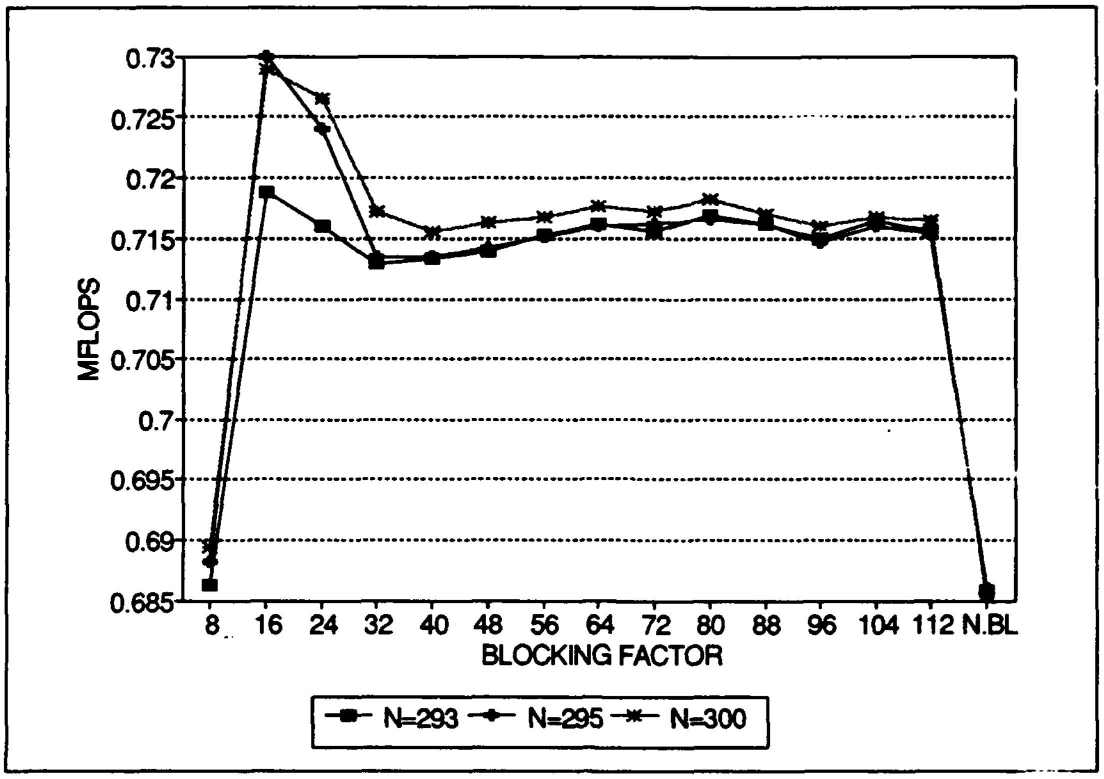

Figure 20

The Performance of Blocked Matrix Multiplication on Solbourne S4000 SPARC. Compiler optimization parameters are not enabled during compilation process. The abbreviation N.BL. means No BLocking

and most of them were to access memory, implying its efficiency.

Figure 21 is similar to Figure 20 except that *gcc* compiler optimization was invoked. The code was compiled by the command "cc **-O2** source_code.c " at the prompt. The performance on Solbourne S4000 SPARC station gets better when the block size is 16. Because it has a 2kB 2-way set-associative data cache and a word length of 4 bytes. This corresponds to a data cache size of 512 words. For a local memory of size *C*, the optimal blocking factor is roughly *sqrt(C)*[9]. Since the data cache in SPARC station is two-way set associative, *sqrt*(512/2) is equal to 16 and therefore a block size of 16 x 16 results in a better performance. As observed in Figure 21, optimized blocked code with a blocking factor of 16 is 40% better in performance than the one with no blocking.

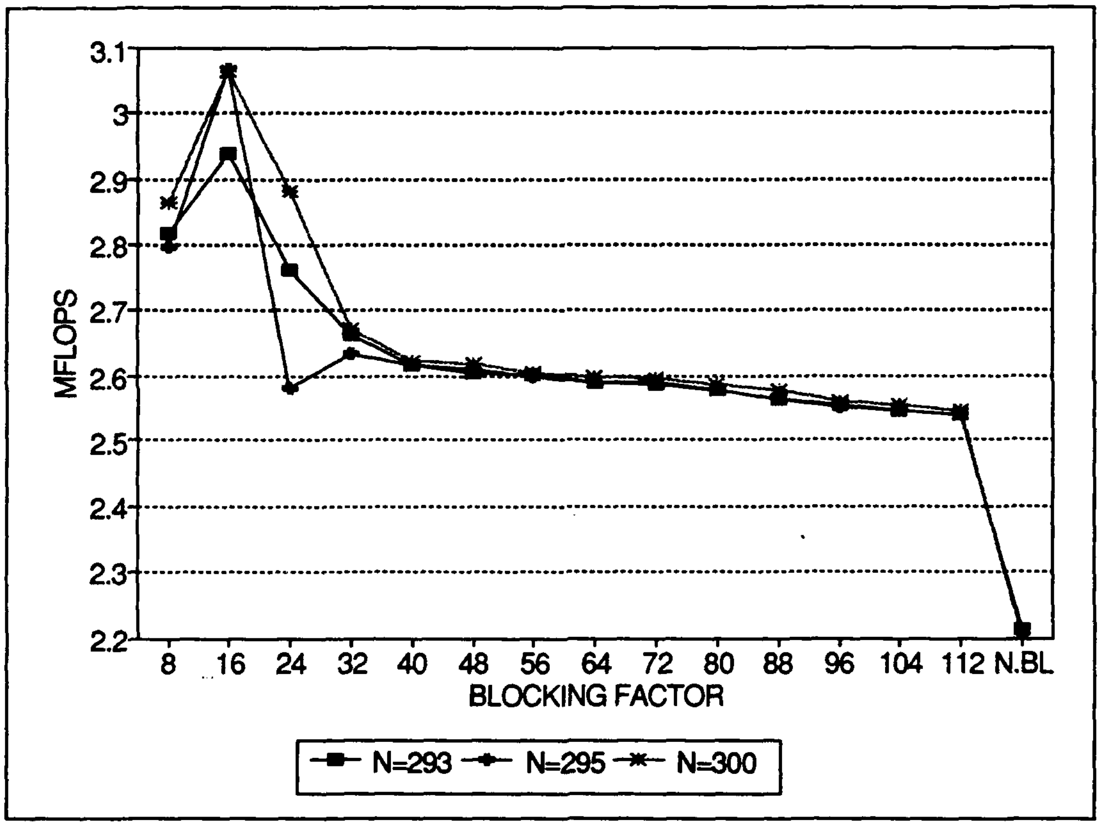

Figure 21

The Performance of Blocked Matrix Multiplication on Solbourne S4000 SPARC station. Compiler optimization switch -O2 is enabled

b. *Cache Performance Experiments with Blocked Codes on DECstation 3100*

Figure 22 shows the graph for the compiler optimized blocked code. Blocking optimizes the code and we also observe a better performance of blocking if the code is compiler optimized for DECstation 3100. As shown in Figure 22, matrix sizes 295 and 300 present almost the same performance of the data cache and consistent increase until we reach block size of 72. At block size of 80, both of them show their highest performances. After this block size, no big change is observed in the plottings of these two matrix sizes. Matrix size 293 present a steadily increasing performance like the previous two. But, after achieving its best performance at block size of 40, it behaves differently and the performance for the data cache degrades drastically beginning from the block size of 56. The lowest performance for three matrix sizes is obtained when no blocking is applied. Because, the interference misses of the cache increases in this case. This means that the data which we can reuse are replaced by self-interfering array elements.

c. *Comparison of The Two Machines with respect to Speedup measure*

The two machines, DECstation 3100 and Solbourne S4000 SPARC station are compared with respect to speedup for different blocking factors in Figure 23. *Speedup* is calculated by dividing each performance value by performance of unblocked code for the respective machine. Solbourne S4000

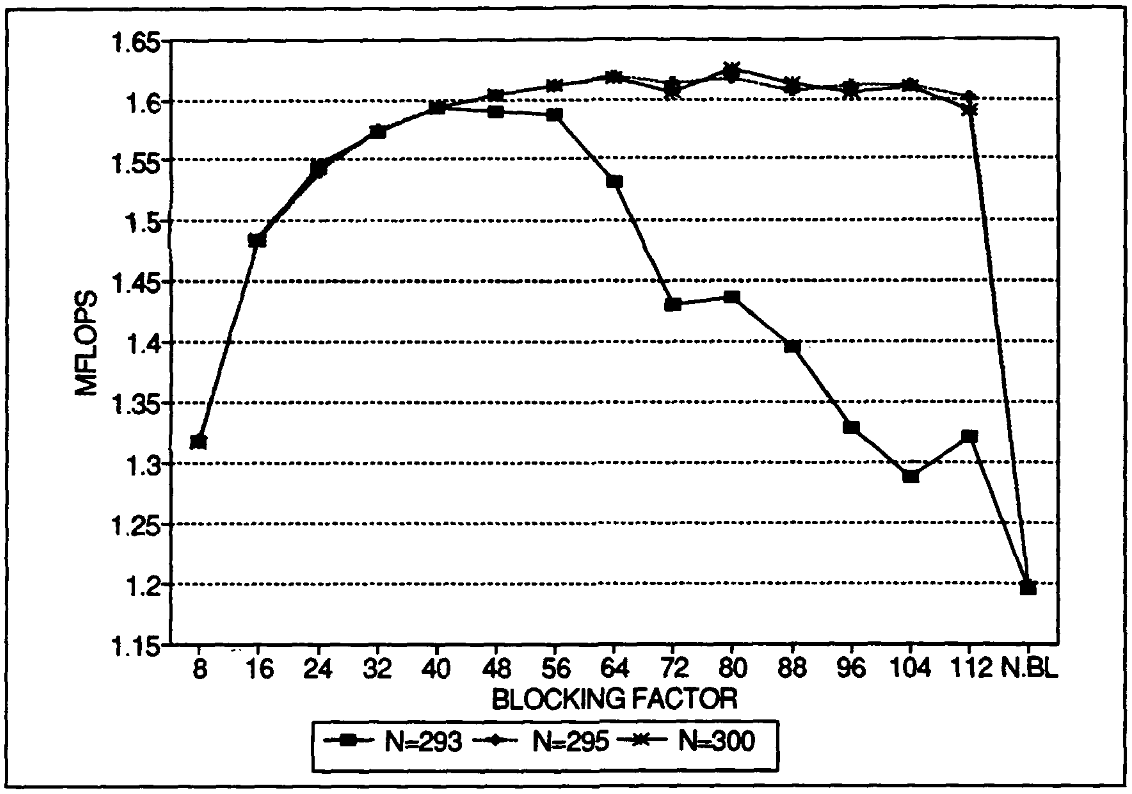

Figure 22

The Performance of Compiler Optimized Blocked Matrix Multiplication on DECstation 3100. Compiler optimization switch -O2 is used for compilation process

SPARC station shows the highest increase in performance at block size of 16 for three matrix sizes. This is related to the data cache size. But this increase does not last long and a decrease is observed after block size 16 which is the optimal blocking factor for SPARC station. The speedup decreases consistently after blocking factor of 32. As the block size increases over this value, the performance of the data cache decreases due to self-interference.

For matrix sizes 295 and 300, DECstation 3100 shows consistent performance increase as the blocking factor

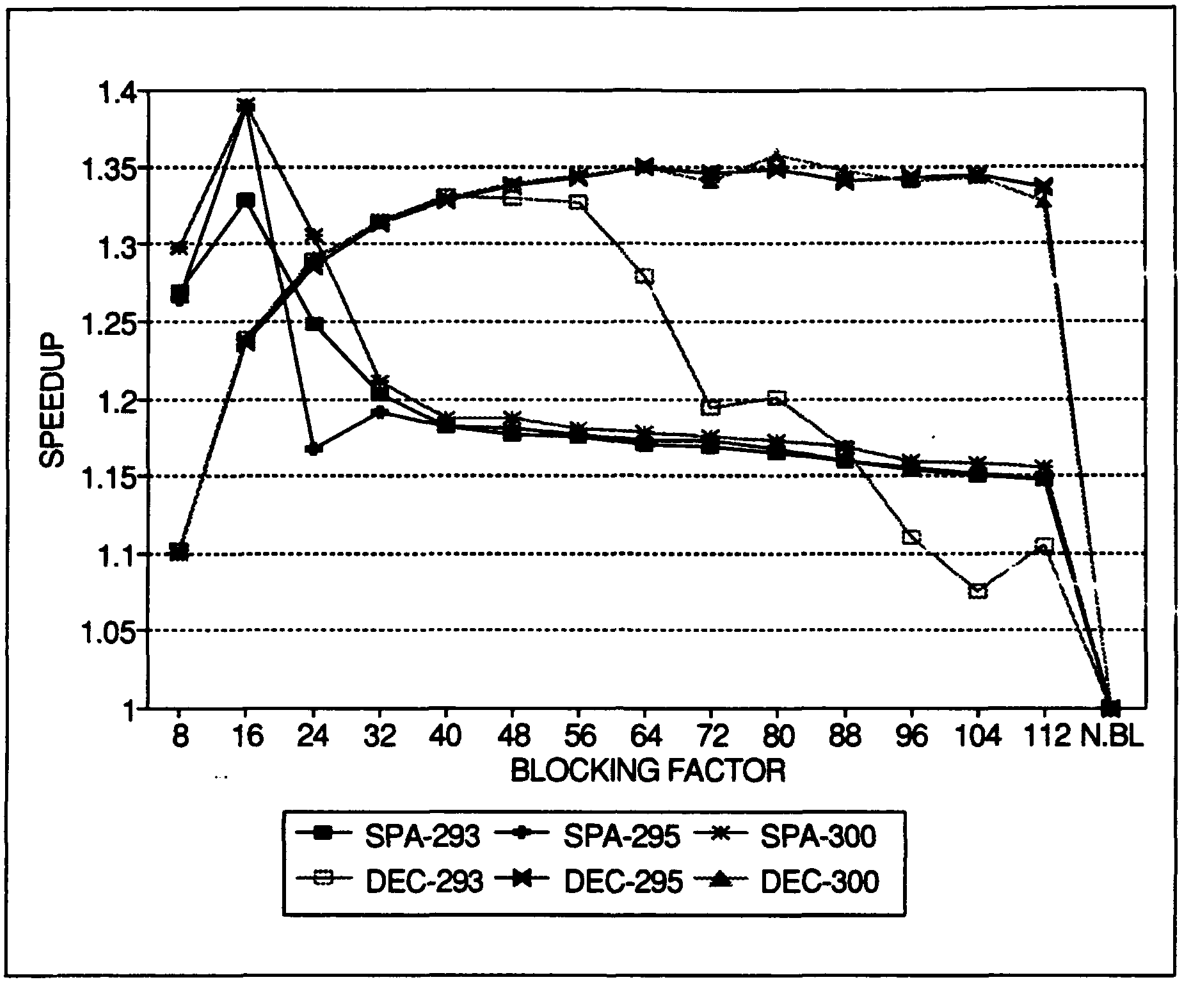

Figure 23

Comparison of DECstation 3100 and Solbourne S4000 SPARC station with respect to speedup vs blocking factors for matrix sizes of 293, 295, 300

increases. But, matrix size of 293 does not behave like the previous two. The performance of data cache gets worse after the blocking factor of 56. This shows that very similar matrix sizes may behave differently. Data cache in DECstation 3100 displays better performance for a wide range of blocking factors where the data cache in Solbourne S4000 SPARC station performs its best just for blocking factor of 16.

d. *Cache Performance Experiments with Blocked Unfolded Codes on Solbourne S4000 SPARC station*

In an attempt to furthermore increase in the performance of the data cache, experiments with *loop unfolding (unrolling)* technique were performed. Loop unfolding is the process of replacing the iterations of a loop with noniterated straight-line code [Ref. 16]. Shown in Example 4 are the codes for matrix multiplication without and with any unfolding. The code is unfolded 3 times, where U is the variable to show how many times the loop is unfolded.

Example 4.

```
  for (k=kk; k<min(N,kk+B); ++k) {
      temp = a[i][k];
      M  = min(N,jj+B);
      for (j=jj; j<M; j=jj+B) {  /* no unfolding */
          c[i][j] +=  temp * b[k][j];
      }
```

is unfolded to

```
 for (k=kk; k<min(N,kk+B); ++k) {
     temp = a[i][k];
     M  = min(N,jj+B);
     if (M % U==0) {  /* control statement for unfolding */
        for (j=jj; j<M; j=j+U) { /* U=3, unfolded 3 times */
```

```
            c[i][j]   +=  temp * b[k][j];
            c[i][j+1] +=  temp * b[k][j+1];
            c[i][j+2] +=  temp * b[k][j+2];
        }
    }
        else {
          for (j=jj; j<M; ++j) {
              c[i][j] +=  temp * b[k][j];
          }
        }
```

The data obtained from the experiments with unfolded blocked matrix multiplication code shows that the overhead on the control statement plays an important role in reducing the performance. So, blocked codes with and without unfolding achieve negligibly different performances. Figure 24 illustrates this cache behavior for the unfolded 8 times. Unfolding did not improve the performance on the blocked code because of the control statement overhead. The degree of unfolding the loop iterations affects the performance slightly.

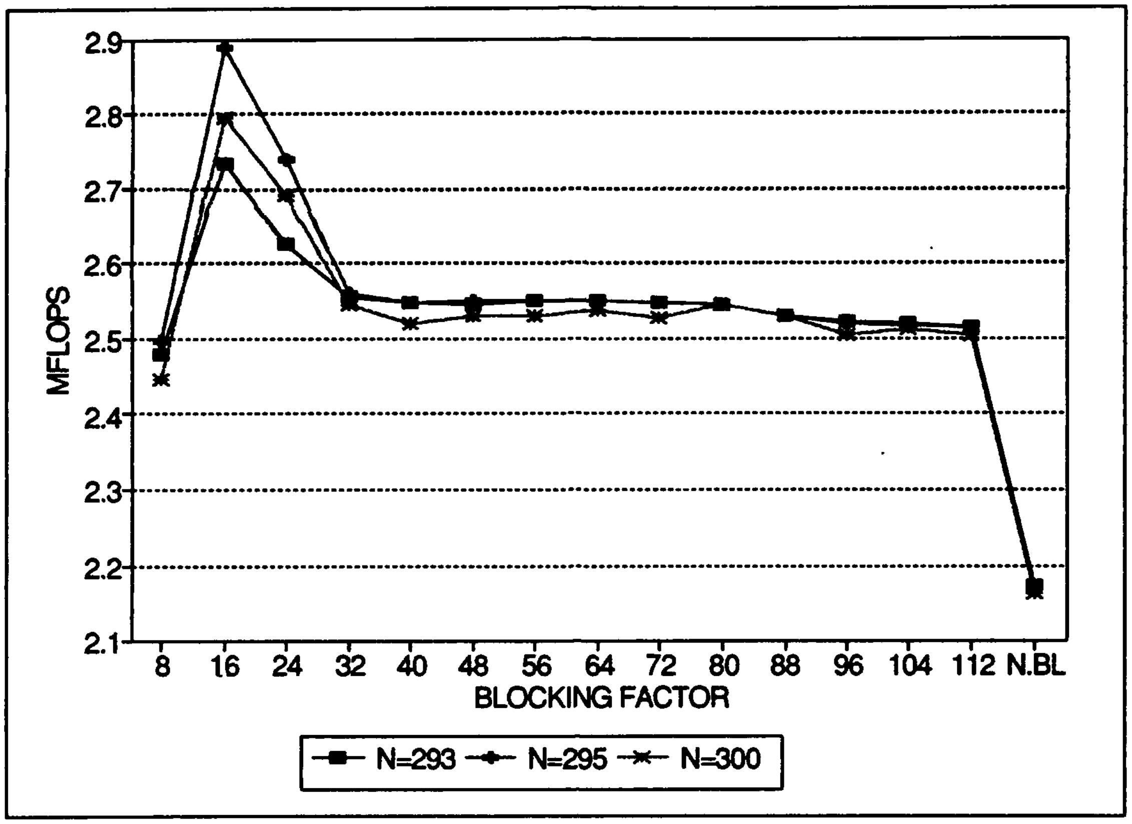

Figure 24

Performance for a blocked and 8 times unfolded matrix multiplication code, Solbourne S4000 SPARC station

The graph in Figure 25 plots the performance levels for blocked matrix multiplication code with 16 and 8 times unfolded. As presented in Figure 25, the performances of 16 and 8 times unfolded codes are not extremely different from each other. This again shows that the control statement (if statement) in the code before the unfolded lines of loop iterations is the main reason of the overhead, rather than the iterations which are not unfolded.

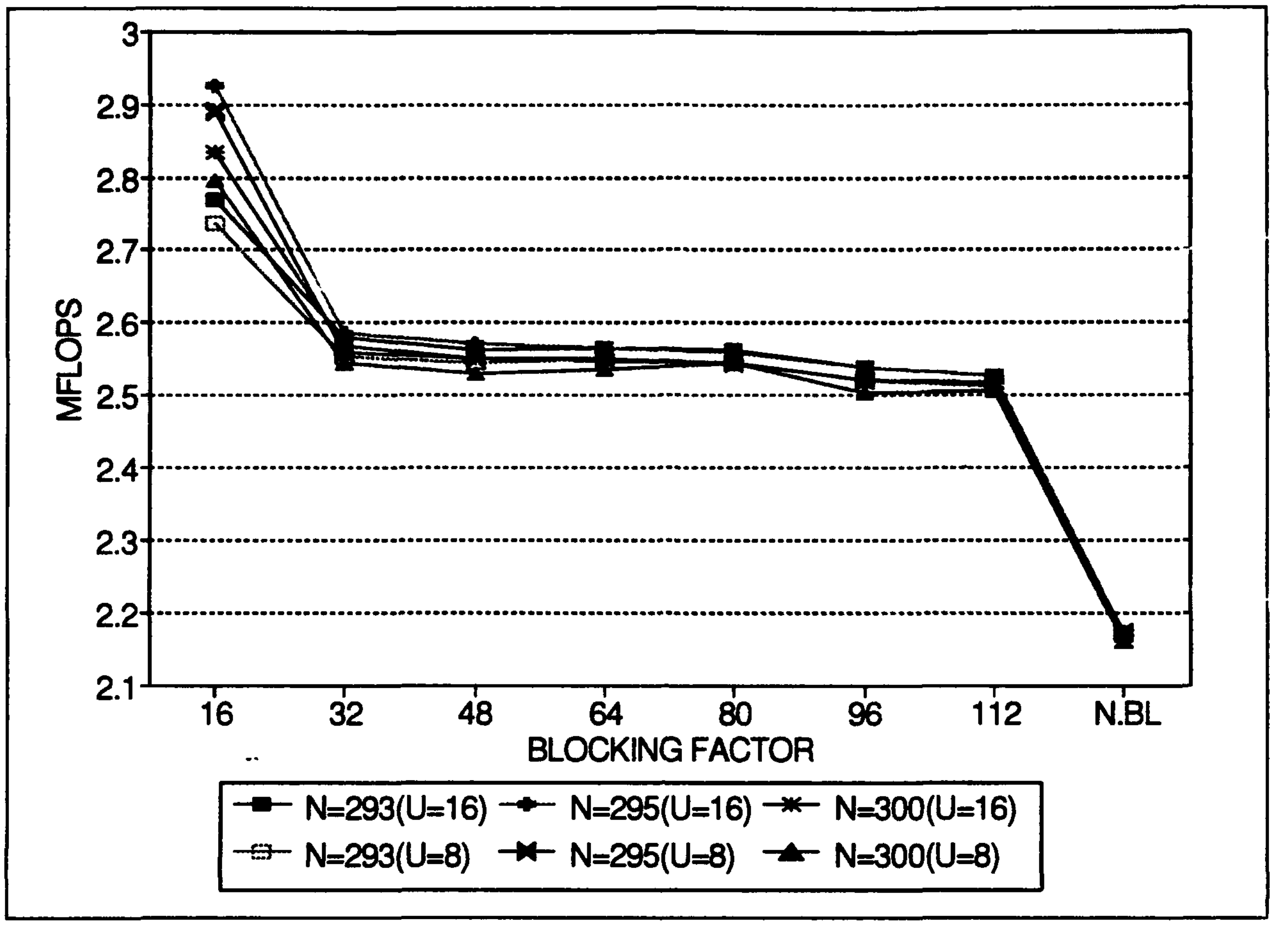

Figure 25

Performance for a blocked matrix multiplication code unfolded 16 and 8 times, Solbourne S4000 SPARC station

III. ALGORITHMS TO IMPROVE DATA CACHE PERFORMANCE

A. SELF-INTERFERENCE IN CACHE

A reused variable of an array will miss in the cache if the references between reuse occupy the same cache location, thus leading to self-interference misses. If the accesses are done in a stride one manner, then no interference will occur unless the number of accessed data is bigger than the cache size. Otherwise, since the access pattern of a stride one or a constant stride array is uniform, its self-interference pattern also becomes uniform and increases the cache misses. In order to avoid self interference totally, "*the largest block size that does not suffer from any self-interference* (B_C)" is computed.

B. THE CRITICAL BLOCKING FACTOR

Lam, Rothberg, and Wolf have developed an algorithm to find the largest square block that avoids self-interference for a given matrix size. This algorithm tailors the blocking factor according to the problem size, i.e., matrix size, to improve the average miss rate [Ref. 9]. By doing so, they intend to improve the average miss rate and to reduce the variance. Since the periodicity in the addressing of a direct-mapped cache and the constant-stride accesses are obvious, it is relatively easy to determine B_C. Their determine the largest square block with no self-interference is shown in Figure 26.

```
algorithm FindB(N, C : integer) return integer;
  maxWidth, addr, di, dj, X, Y, N : integer;
  maxWidth = min(N, C);
  addr  = N/2;
  while true do
    addr = addr + C;
    di   = addr div N;
    dj   = abs((addr mod N) - N/2);
    if (di >= min(maxWidth, dj))
       return min(maxWidth, di);
    else
       maxWidth = di >= min(maxWidth, dj;
  end while;
end algorithm;
```

Figure 26

Lam's Algorithm to compute the Critical Blocking Factor B_c

1. Sensitivity of B_c with respect to Matrix Size

B_c obtained from Lam's algorithm for some matrix sizes draws attention to its sensitivity with respect to matrix size. Figure 27 shows the critical blocking factor for a some matrix sizes between 293 and 323.

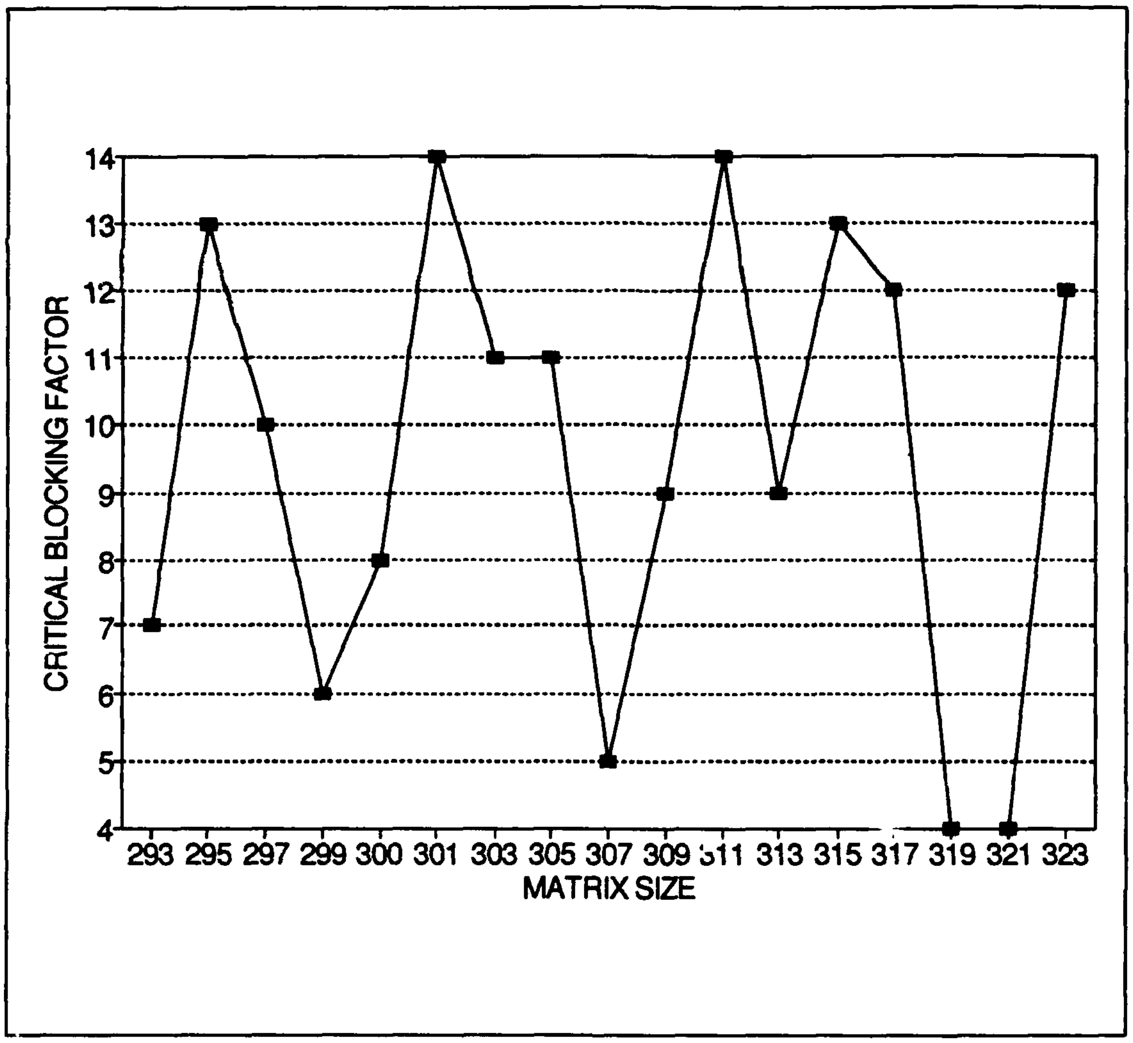

Figure 27

B_c versus Matrix Size. Solbourne S4000 SPARC Station

The critical blocking factor B_c cannot be larger than sqrt(C), so the run time of the algorithm is *O(N/sqrt(C))* [Ref. 9]. Figure 28 plots the performance of the data cache for the values shown in Figure 27. The performance is greatly affected by value of B_c, dropping to 1.7 Mflops for a 321x321

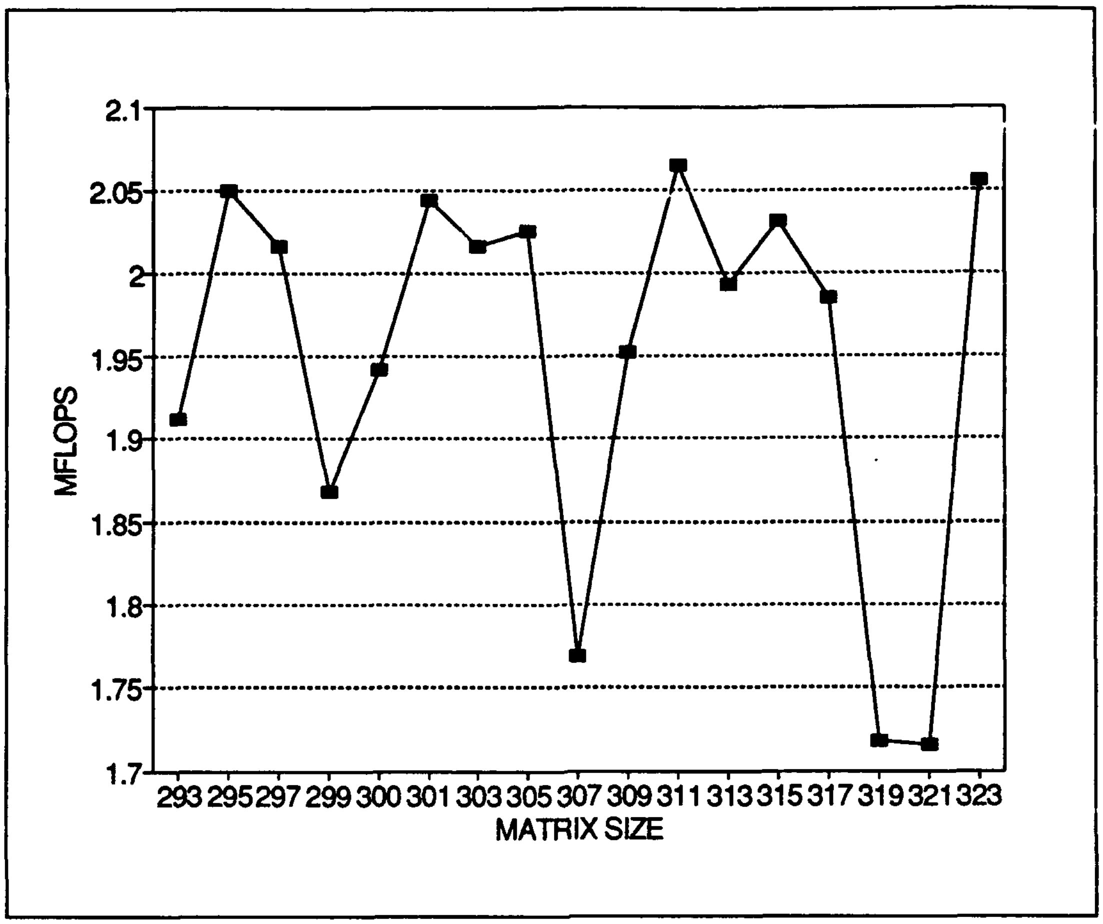

Figure 28

Data Cache Performance versus Matrix Size. B_c is obtained by using Lam's Algorithm for Matrix Size. Solbourne S4000 SPARC Station.

matrix and jumping to 2.0 Mflops (an increase of 17%) for a 323x323 matrix.

2. Effect of Declared Matrix Size on B_c

It is important to obtain consistent performance from an algorithm. Small changes in the values of the inputs generally should not cause huge swings in the performance of the algorithm. Users tend to use arbitrary array sizes larger than maximum expected problem size.

If we use Lam's Algorithm which changes the block size by searching for a better B_c, then, for a given matrix size, we can keep increasing the size of the matrix and determine another B_c until we find a block size which gives a better performance, preferably sqrt(C) where C is an integer.

In Figure 29, the performance variation is shown for actual matrix sizes of 293, 295, and 300, where B_c is computed while extending the size of the matrix. Here, we demonstrate the sensitivity of Lam's Algorithm to the changes on the actual matrix size. Even a small change on one dimension of array b[k][j] affects the performance of the code blocked with a blocking factor for a matrix size as big as that new dimension. For example, when we use actual matrix size of 293 without any change on any dimension, data cache performance is 2.7 Mflops. If we declare one dimension of the same matrix to be 310, then the performance jumps up to 3.05 Mflops (an increase of 13%). For dimension of 321, the performance decreases from 3.05 Mflops to 2.2 (a decrease of 38%).

Since calculation of B_c is sensitive to this small change, we can draw insights to this fact as follows:

1. Block size is highly sensitive to matrix size.
2. Some bigger block sizes after this change shows that cache is not filled as much as possible with the block size computed with the algorithm. In other words, the bigger portion of cache used, the better cache performance for some array sizes. As long as the in the cache, in a certain range, if we increase the block size, it helps decreasing misses. The following equation in

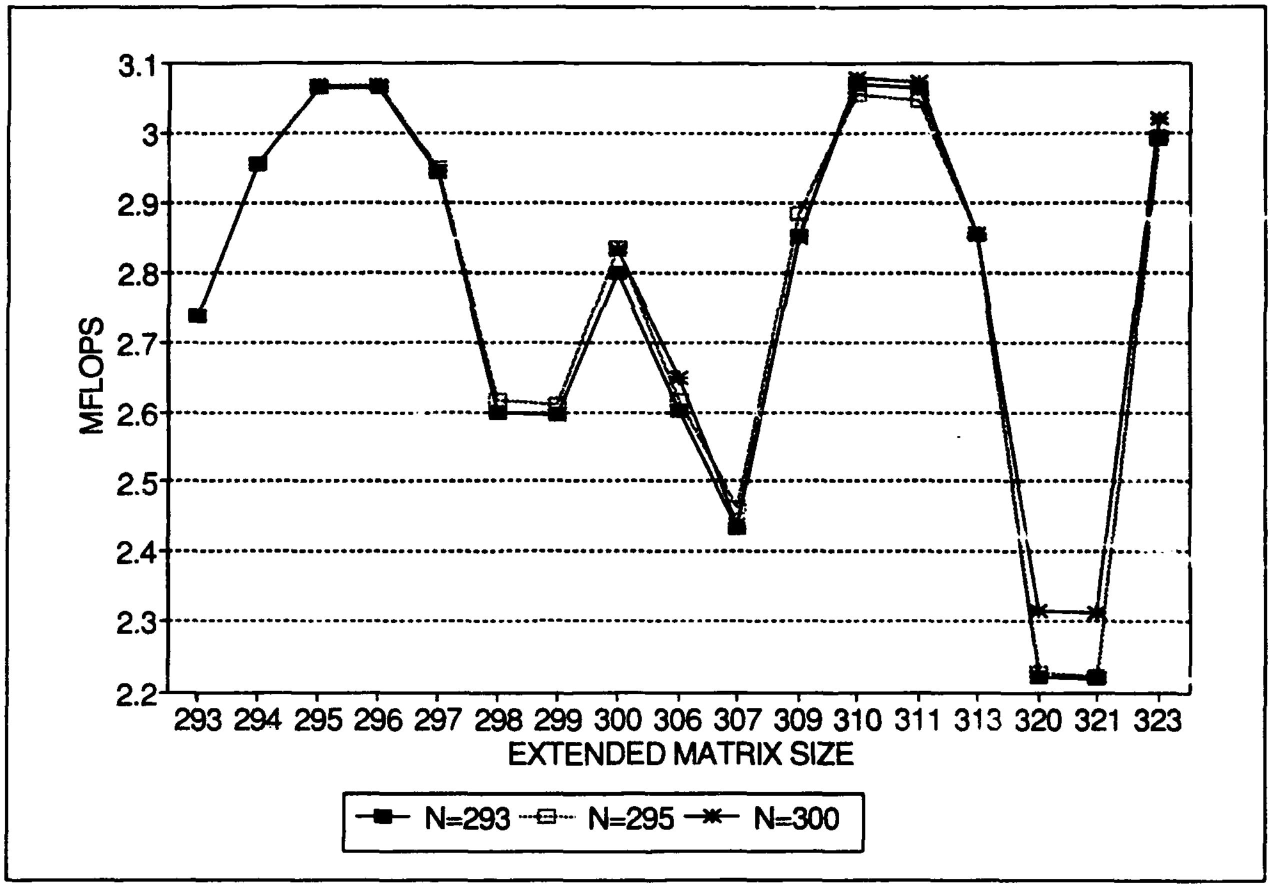

Figure 29

Extended Dimension versus Performance for Actual Matrix Sizes of 293, 295, and 300. Solbourne S4000 Sparc Station

Figure 30 models the miss rate in terms of parameters used in the blocked matrix multiplication and calculates the total number of cache misses [Ref. 9].

$$= N^3[2/B + Si(b) + 3(1 - Si(b))B/C + B/C]$$

where N^3 *= the number of operations performed,*

B = *blocking factor,*

C = *cache size,*

Si(b)= *$(1 - B/C)^{B-1}$ = self-interference of accessing B-1 rows of array b before the same data is reused.*

Figure 30

Equation to compute total number of misses in cache

According to this equation, there are $N^3(2/B)$ compulsory misses, misses that are intrinsic to the algorithm given the blocking factor and cannot be avoided even if the address mapping is perfect. The factor Si(b) is due to self interference among the elements in the b array. Any blocking factor lower than the critical blocking factor does not cause self interference. The other two terms are due to cross interference between different variables. Example 5 below is presented to quantitavely demonstrate the effect of B_C.

Example 5.

If actual matrix size = 293 and C = 256, then B = 7 is computed with Lam's Algorithm presented in Figure 26. With these values, the total number of misses = $1.17N^3$. If we fix the cache size and extend the declared matrix size to 304 as determined with our seacrh technique, then B_C = 16 is computed with these values, the total number of misses = $0.68N^3$. We showed that the total number of misses for a block size obtained with Lam's Algorithm is worse than that of obtained with the extended matrix size (1.17/0.68 = 1.72, 72% degrades the performance). Therefore, having a bigger blocking factor and an extended matrix dimension leads to a better data cache performance, while decreasing the miss rate.

C. CHANGING ARRAY SIZE VIA A SEARCH TECHNIQUE AND DETERMINING THE CRITICAL BLOCKING FACTOR

We present a search technique to find values of B_c larger than those computed using the algorithm in [Ref. 9]. This technique computes B_c by searching through all blocking factors for the range of matrix sizes up to 10% bigger than the actual matrix size. It mutually employs the algorithm in Figure 26 until it returns the biggest blocking factor, less than or equal to sqrt(C), where C is cache size, and bigger than 1, which is the smallest blocking factor that we can assign. We demonstrated that this blocking factor improves data cache performance as shown in Example 5. The following C language code in Figure 31 illustrates this technique.

```
/* algorithm COMPUTE_B */
#define min(a,b) ((a<b) ? a:b) /* min function */
int algo(int, double);
main()
{
  int k, N, Nmax, B, Bmax=1, i, Nlimit;
  double C;
  printf(" Enter the size of the matrix:\n");
  scanf("%d", &N);
  printf(" Enter the size of the cache:\n");
  scanf("%d", &C);
  Nmax = N;
  k = N * 0.10;    /* max allowable matrix size increase */
  Nlimit = N + k;  /* max matrix size */
  for (i = N; i <= Nlimit; ++i) {
     B = algo(N, C);
     if ((B <= sqrt(C)) && (Bmax < B)) {
        Bmax = B;
        Nmax = N;
     }
  }
  printf("New Matrix Size= %d\n", Nmax);
  printf("Blocking Factor= %d\n", Bmax);
}
  int algo(N, C) /* Lam's Algorithm (algorithm FindB)
                    in Figure 1 */
  .....
```

Figure 31

A search technique coupled with Lam's Algorithm to find B_C

The technique keeps extending the actual matrix size within the limit of 10% of the actual matrix size. For every size, it computes B_C by repetitively applying Lam's Algorithm in

Figure 26, and retrieves the first biggest blocking factor and the corresponding extended matrix size.

Here the choice of 10% is arbitrary. In practice, we do not need to extend both rows and columns of the actual matrix. Because, the advantage of this approach is that it finds an extended matrix size which results in a better data cache performance than that of Lam's Algorithm. Miss rates obtained from the equation in Figure 30 by substituting the values for two different matrix sizes show that matrix size and blocking factor from our algorithm present a better total number of misses. Moreover, in Example 5, cache portion used for the blocking factor computed with Lam's Algorithm is

(7 x 7) / 256 = 0.19 --> 19% of the cache,

while the blocking factor computed with our technique allows

(16 x 16) / 256 = 1.00 --> 100% of the cache to be utilized.

Therefore, the technique presented above als[illegible]es us to utilize data cache optimally. However, this approach suffers from using additional columns/rows of a matrix.

D. A NEW ALGORITHM TO FIND THE CRITICAL BLOCKING FACTOR WITH NO SELF-INTERFERENCE

The drawback of the search technique in the previous section is that it uses Lam's algorithm which has a complexity of *O(N/sqrt(C))* [Ref. 9]. For each extended matrix size, using this algorithm becomes an overhead for the search technique. Additionally, we have no guarantee for an acceptable choice of B_c.

In this section, we introduce a new algorithm which improves data cache performance by determining the critical blocking factor with no self-interference. The algorithm is shown in Figure 32.

```
/* DIRECT Algorithm */
int gcd(int, int);
main()
{
  int SIZE, C, C', OLD_SIZE, B=1, X, GCD;
  OLD_SIZE = SIZE;
  if (sqrt(C) is not an integer )
     C' = C/2;
  else
     C' = C;
  X = sqrt(C)-(SIZE % sqrt(C'));/* these two lines set
                                   matrix  */
  SIZE = SIZE + X;              /* to be a multiple of
                                   sqrt(C) */
  do {
    GCD = gcd((SIZE/sqrt(C')), sqrt(C'));   /* gcd test */
    if (GCD = 1)
       B = sqrt(C');
    else
       SIZE = SIZE + sqrt(C');
    } while (B /= sqrt(C'));
    gcd(int V1, int V2) {
      int temp;
      while (V2) {
          temp = V2;
          V2 = V1 % V2;
          V1 = temp;
      }
      return V1;
    }
  }
```

Figure 32

A New Algorithm to find B_c

Direct Algorithm allows us to use sqrt(C) as optimal blocking factor if sqrt(C) is an integer all the time. If not, the optimal blocking factor is considered as sqrt(C/2), because we obtain a multiple of two for C/2. The underlying principle is to cover data cache as much as possible at the expense of potentially using more memory.

We assume that the matrix has sqrt(C) elements, where C is the cache size. If each element in the matrix, which we call it *super_ element E*, contains sqrt(C) elements with *a stride S* between them and if cache is divided by the size of sqrt(C) super_elements, then *(C / E)* corresponds to the number of blocks in the cache. So, *if S and (C / E) are relatively* *prime,* which also means that *their greatest common divisor (gcd) is 1,* then every super_element will be placed in a new *super_location* up to (C / E) different super_locations.

The advantages of Direct Algorithm:

- The time consumed to compute the extended matrix size and the corresponding B_C is shorter than the one in Lam's Algorithm. Therefore, we save time.
- The blocking factor is always sqrt(C) or sqrt(C/2) due to the value of cache size, at which we get a better performance than that of Lam's Algorithm.

The disadvantages of Direct Algorithm:

- As in the previous section, this algorithm also suffers from using additional rows/columns which is an overhead for the computation. The maximum addition in size is 2 * SQRT(C) -1 (if SQRT(C) is an integer).

This algorithm is experimented on Solbourne S4000 SPARC station and DECstation 3100. Figure 33 plots the performance of the data cache for a wide range of matrix sizes on SPARC station. Figure 34 plots the actual matrix size versus the extended matrix size obtained with Direct Algorithm.

This algorithm finds a matrix size suitable to be used with a block size of sqrt(C) if it is an integer (or sqrt(C/2) if not). Since cache size is 512 words for a two-way associative data cache in SPARC station, sqrt(512/2) = sqrt(256) =16 is an integer. So, we use almost 100% of the cache. The above performance curve in the graph shows the performance for a matrix size determined by *Direct Algorithm* with a block size of sqrt(C). It presents a better performance level compared

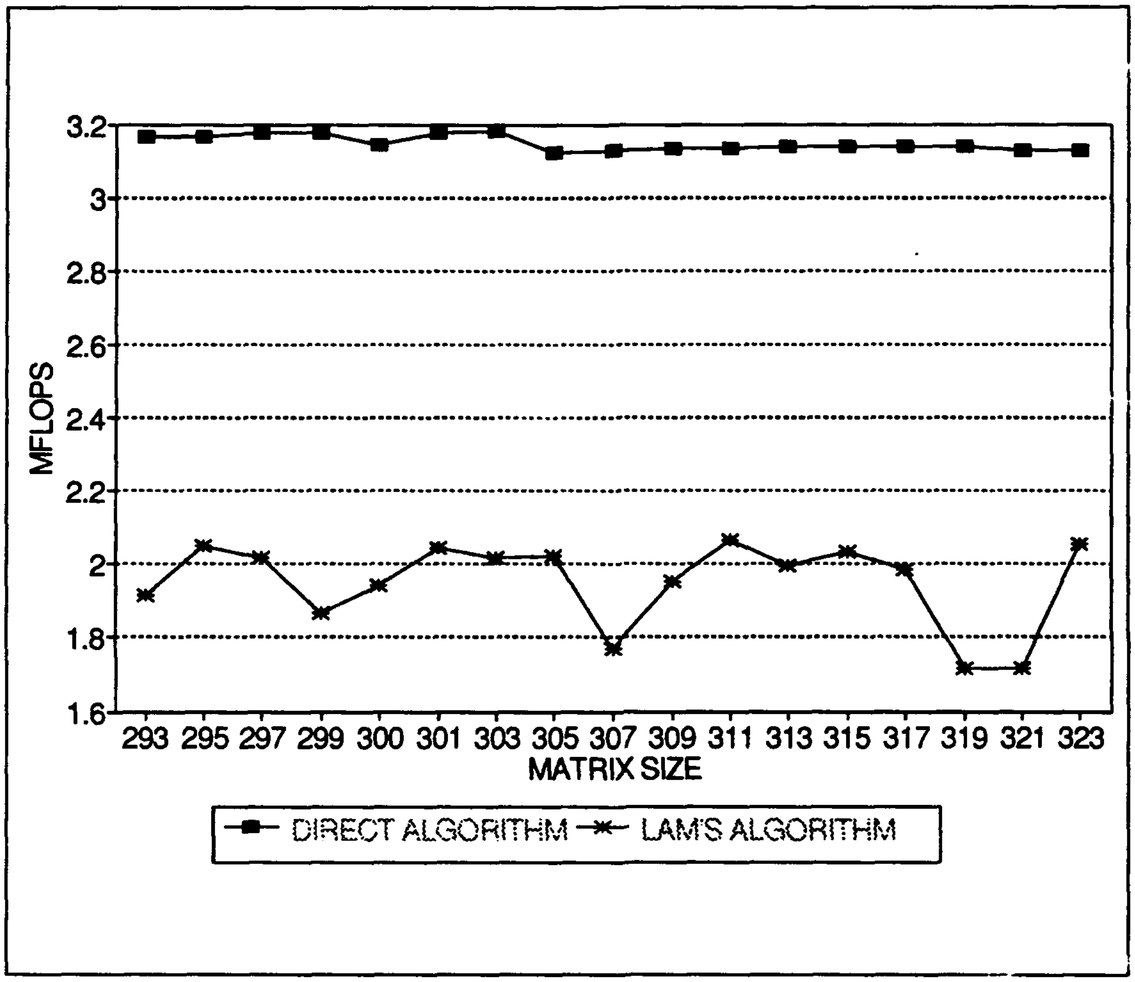

Figure 33

Performance levels with optimal block size of 16 (without any change on matrix size) and with a block size of sqrt(C) (for extended matrix size determined by the algorithm above) on the data cache of Solbourne S4000 SPARC station.

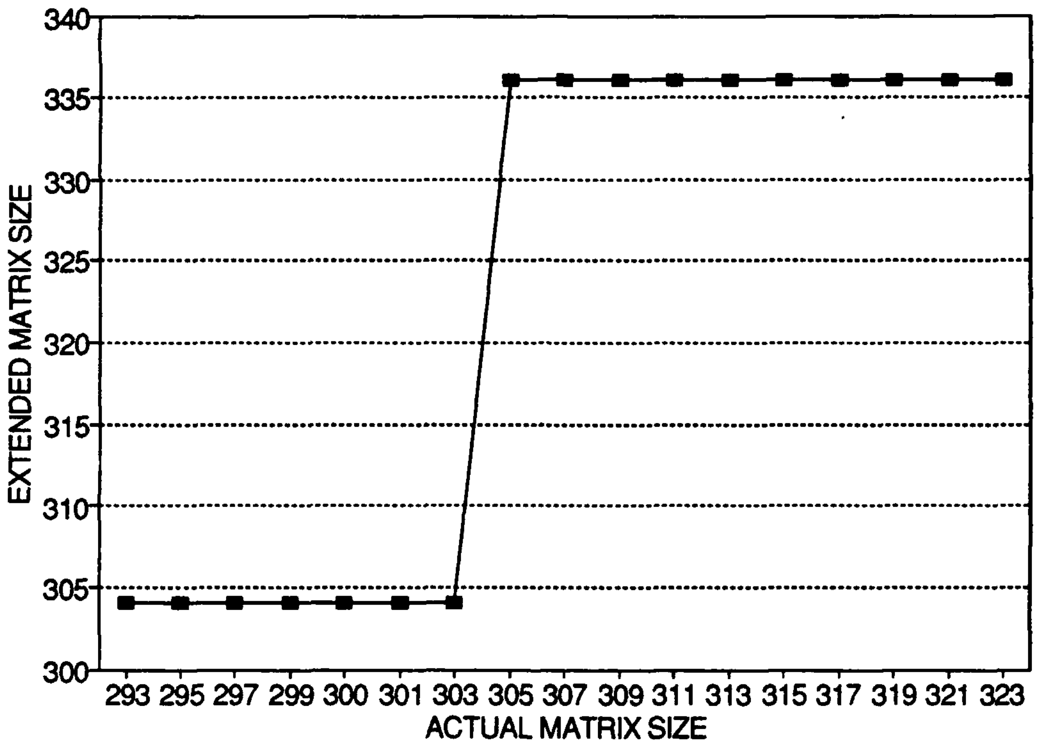

Figure 34

The Actual Matrix Size versus the Extended Matrix Size obtained with Direct Algorithm. Solbourne S4000 SPARC station

to the one below and behaves very consistent. The variation in performance for the one below is due to the interference misses in the data cache.

In the experiment on DECstation 3100, three matrix sizes(293, 295. 300) are used and performance level of the data cache is observed for the values obtained with the new algorithm and the best blocking factors obtained previously.

Figure 35 shows the two performance levels of the data cache on DECstation 3100.

Figure 36 plots the actual matrix size versus the extended matrix size obtained with Direct Algorithm. DECstation 3100 has 8K double word direct mapped cache where sqrt(C) is not an integer. So, Direct Algorithm gets half of the cache and determines the new matrix size shown in Figure 36 for sqrt (C/2).

Our algorithm performs its best if the number of distinct array references to data cache is less than or equal to the associativity of data cache. Otherwise, interference among the references may occur.

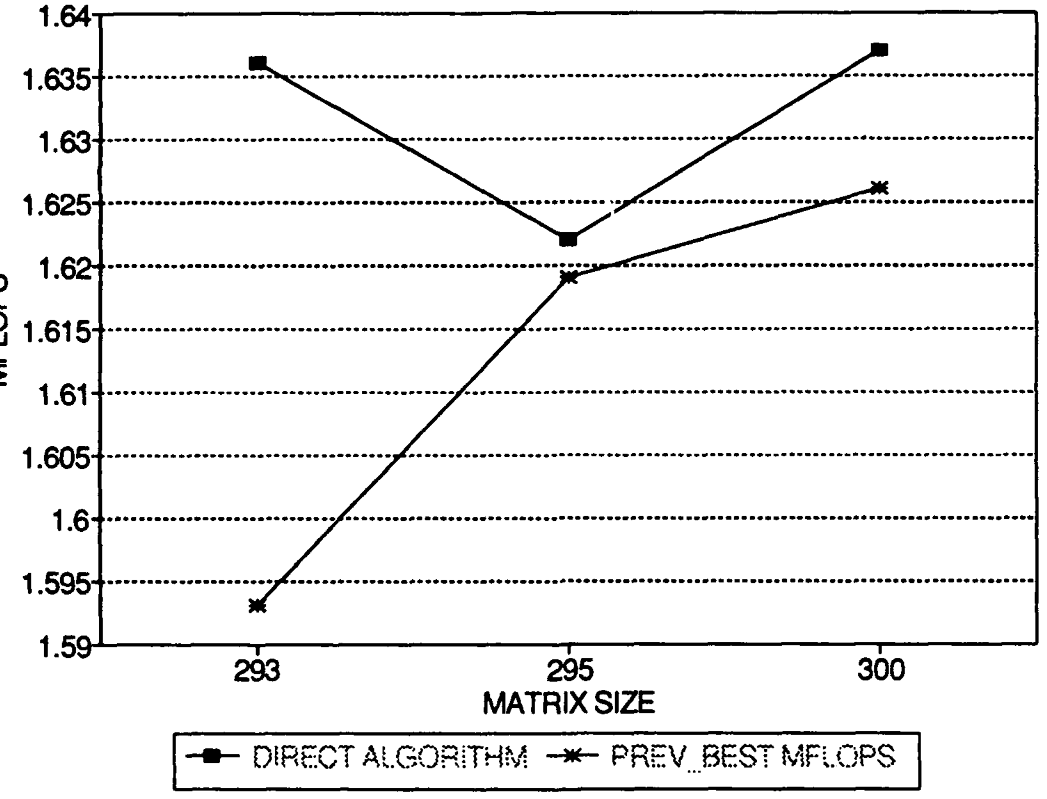

Figure 35

Performance levels with optimal block sizes (with no change on matrix size) and with a block size of sqrt(C/2) (with matrix size determined by the algorithm above) for the data cache on DECstation 3100

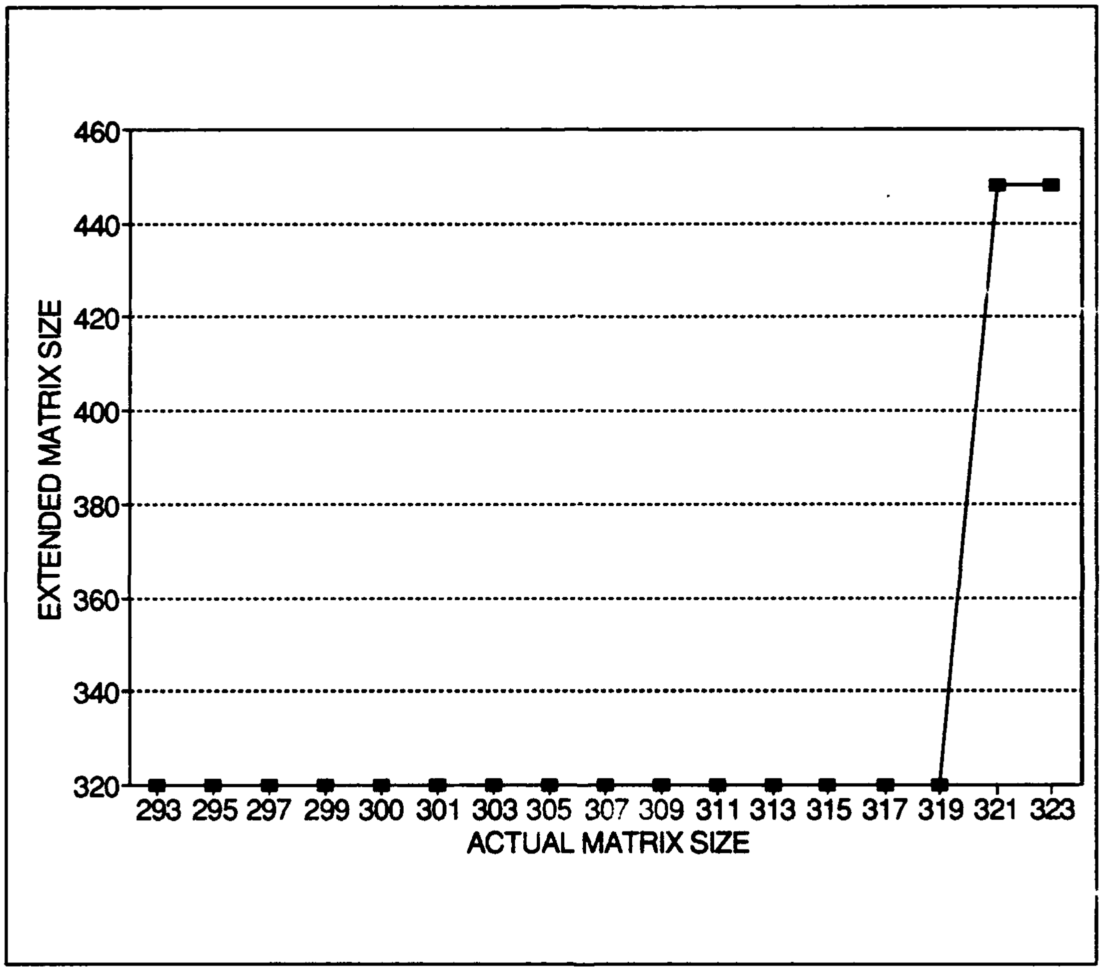

Figure 36
The Actual Matrix Size versus the Extended Matrix Size obtained with Direct Algorithm. DECstation 3100

IV. CONCLUSIONS AND RECOMMENDATIONS FOR FURTHER RESEARCH

In Chapter I, we reviewed the notion of reuse and reported the importance of the number of cache misses for RISC processors. Since there is an amazingly large reuse of data in cache, we focused on some techniques, namely **blocking techniques**, which are used to reduce memory traffic and exploit this data reuse.

In Chapter II, we performed several experiments on blocking and presented the results similar to those in [Ref. 9] for a better understanding of the effect of blocking on the performance of two different data caches.

In Chapter III, we demonstrated the sensitivity of the algorithm in [Ref. 9] and the sensitivity of blocking factor obtained by that algorithm to the declared array size. In fact, this sensitivity shows that the performance of the algorithm is very unstable related to the size of the matrix. To remedy this problem, we introduced two algorithms: The first one is *a search technique* which finds values of B_c larger than those computed using the algorithm in [Ref. 9]. This technique computes B_c by searching through all blocking factors for the range of matrix sizes up to 10% bigger than the actual matrix size. It iteratively employs the algorithm in [9] until it returns the biggest blocking factor, less than or equal to sqrt(C), where C is cache size, and bigger than 1, which is the smallest blocking factor that we can assign. The other is *Direct Algorithm* which improves the performance of workstations in some specific applications. This algorithm finds a matrix size suitable to be used with a block size of sqrt(C) if it is an integer (or sqrt(C/2) if not). It gives the best results when the number of distinct array references is less than or equal to the associativity of data cache.

For further research we recommend looking at the algorithms in situations where the number of distinct forms of array references is greater than the cache associativity factor.

LIST OF REFERENCES

1. Walid Abdul-Karim Abu-Sufah. Improving The Performance of Virtual Memory Computers, Ph.D. Thesis, Dept. of Comp. Sci. Rpt. No. 78-945, Univ. of Illinois, Urbana, IL, Nov 1978.

2. W. A. Abu-Sufah, D. J. Kuck and D. H. Lawrie. On The Performance Enhancement of Paging Systems Through Program Analysis and Transformations, IEEE Trans. on Computers, Vol. C-30, No. 5, pp. 341-356, May 1981.

3. Michael Wolfe. Iteration Space Tiling for Memory Hierarchies, Proc. of the 3rd SIAM Conf. on Parallel Processing for Scientific Computing, Garry Rodriguez, Society for Industrial and Applied Mathematics, Philadelphia, PA, pp. 357-361, 1987.

4. R. Irigoin and R. Triolet. Supernode Partitioning, Conf. Record of the 15th Annual ACM Symp. on Principles of Languages, pp. 319-329, Jan. 13-15, San Diego, CA, ACM Press, New York, 1988.

5. K. Gallivan, W. Jalby, U. Meier, and A. Sameh. The Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design. Technical Report UIUCSRD 625, University of Illinois, 1987.

6. D. Gannon, W. Jalby, and K. Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation, Journal of Parallel and Distributed Computing, 5:587-616, 1988.

7. A. Porterfield. Software Methods for Improvement of Cache Performance on Supercomputer Applications, pp.:iii, 4-7, 19-20, 76-77, 100-103, Ph.D. Thesis, Rice University, May 1989.

8. M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm, June 26-28, Toronto, Ontario, Canada, ACM Press, 1991.

9. M. Lam, E. R. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms, Architectural Support for Programming Languages and Operating Systems, Santa Clara, CA, April 8-11, 1991.

10. J.-W. Hong and H. T. Kung. I/O Complexity: The Red-Blue Pebble Game, Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, pp. 326-333, ACM SIGACT, May 1981.

11. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A Set of Level 3 Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software, pp. 1-17, March 1990.

12. A. C. McKeller and E. G. Coffman. The Organization of Matrices and Matrix Operations in a Paged Multiprogramming Environment, CACM, 12(3):153-165, 1969.

13. M. J. Wolfe. More Iteration Space Tiling, Supercomputing '89, Nov 1989.

14. J. Dongarra, L. S. Duff, Danny C. Sorensen, and H.van der Vorst. Solving Linear Systems on Vector and Shared Memory Computers, SIAM Press, 1991.

15. J. M. Levesque, J. W. Williamson. A Guidebook to Fortran on Supercomputers, Academic Press, San Diego, 1989.

16. T. G. Lewis, H. El-Rewini. Introduction to Parallel Computing, pp.:xi, 283, 313-315, Prentice Hall, 1992.

17. H. Zima with Barbara Chapman. Supercompilers for Parallel and Vector Computers, ACM Press, 1991.

18. J. L. Hennessy, D. A. Patterson. Computer Architecture A Quantitative Approach, pp. 11-18, 405-422, Morgan Kaufmann Publishers Inc., San Mateo, CA, 1990.

19. G. Langholz, J. Francioni, A. Kandel. Elements of Computer Organization, pp. 196-197, Prentice Hall, 1989.

20. D. Gannon, W. Jalby. The Influence of Memory Hierarchy on Algorithm Organization Programming FFT's on a Vector Multiprocessor, University of Illinois, 1986.

21. Michael Wolfe. Optimizing Supercompilers for Supercomputers, pp. 6-18, 119-123, The MIT Press, 1989.

22. J. Ramanujam, P. Sadayappan. Tiling of Iteration Spaces for Multicomputers, Proceedings of the 1990 International Conference on Parallel Processing, August 13-17, 1990.

23. R. Schreiber, J. J. Dongarra. Automatic Blocking of Nested Loops, RIACS TR 9038, August, 1990.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Amr Zaky, Code CS/ZA Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	10
4.	Mantak Shing, Code CS/Sh Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
5.	Robert B. McGee, Code CS/Mz Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
6.	Dz. K. K. ligi Personel Egitim Daire Bsk. ligi Ankara, Turkey 06100	1
7.	Deniz Harp Okulu K. ligi Tuzla / Istanbul, Turkey 81700	2
8.	Golcuk Tersanesi K. ligi Golcuk / Kocaeli, Turkey 41650	2
9.	Taskizak Tersanesi K. ligi Kasimpasa / Istanbul, Turkey	2
10.	Dr. Elan Moritz (Code 10T) Coastal Systems Station Naval Surface Warfare Center Panama City, Florida 32407	1

11.	Bogazici Üniversitesi Muhendislik Fakultesi Bilgisayar Bilimler Ana Bilim Dali Bsk. ligi Rumeli Hisari / Istanbul, Turkey	1
12.	Orta Dogu Teknik Üniversitesi (ODTU) Muhendislik Fakultesi Bilgisayar Bilimler Ana Bilim Dali Bsk. ligi Besevler / Ankara, Turkey	1
13.	Hacettepe Üniversitesi Muhendislik Fakultesi Bilgisayar Bilimler Ana Bilim Dali Bsk. ligi Beytepe / Ankara, Turkey	1
14.	Marmara Üniversitesi Muhendislik Fakultesi Bilgisayar Bilimler Ana Bilim Dali Bsk. ligi Goztepe / Istanbul, Turkey	1
15.	Istanbul Teknik Üniversitesi Muhendislik Fakultesi Bilgisayar Bilimler Ana Bilim Dali Bsk. ligi Maslak / Istanbul, Turkey	1
16.	Lieutenant Junior Grade Atilla Demirhan Dz. K. K. ligi Personel Egitim Daire Bsk. ligi Ankara, Turkey 06100	2