DATA-CENTRIC COMPILATION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Induprakas Kodukula

August 1998

DATA-CENTRIC COMPILATION

Induprakas Kodukula, Ph.D.

Cornell University 1998

This dissertation presents a novel compiling strategy called *data-centric compilation* for generating highly optimized implementations of programs on modern high-performance computers. The goal of data-centric compiling is to optimize programs for machines with memory hierarchies by reducing the flow of data between different levels of this hierarchy. In addition, it can be used to parallelize semi-structured applications. Experiments on Octane workstations from Silicon Graphics are used to validate this approach in practice.

These results have persuaded a major workstation vendor to adopt data-centric compilation in their compiler product line.

Biographical Sketch

Induprakas Kodukula was born in India. He graduated from the Indian Institute of Technology at Madras with a B.Tech in Computer Science in June 1992. He joined the Ph.D program in the Department of Computer Science at Cornell University in August 1992. He received the Cornell Graduate Field Award as the best incoming student that year.

He received an M.S. in Computer Science from Cornell University in 1995 and a Ph.D in 1998. His area of specialization is High Performance Compilers. He has a minor in Scientific Visualization.

This dissertation is dedicated to my wife, Veena.

Acknowledgements

This dissertation would not have been possible without my advisor, Keshav Pingali. It has been a pleasure and a privilege working with him, and I owe him much of the way I think today. I owe him a special word of gratitude for his support and encouragement throughout the period I was making my career choices. Thorsten von Eicken has made many valuable suggestions to improve this dissertation. I'm grateful to Bruce Land for serving as my minor advisor and Sam Toueg for serving on my committee.

Helene Croft has been a tremendous source of administrative support throughout my PhD, from making travel arrangements to typesetting parts of this dissertation. There have been innumerable occasions where she has kept company through the very last minute while I was trying to meet paper submission deadlines.

I would like to thank Dror Maydan for providing me an opportunity to transfer my research to a production setting. I have enjoyed working with Robert Cox. He patiently answered all my questions about the inner workings of the compiler. I had many fruitful discussions about loop transformations, locality enhancing transformations and other things with Rob Schreiber, who also introduced me to Piazza's

Fine Foods in Palo Alto and their great hummus-on-black-rye sandwich.

My stay in Ithaca was greatly enriched by my roommates, particularly, Badri(1), Badri(2), and Harish. I will miss them all. Vladimir has been a great friend and colleague. Nawaaz, Paul and Vijay have been good company at the office.

I am grateful to my parents for having provided me the opportunity to come to Cornell University for pursuing the PhD program. Without their support, I would never have left India in the first place. I am indebted to my parents and my brother for their love and guidance. Words are not adequate to describe my feelings towards my wife, Veena. Her love, support and dedication have been invaluable, especially during the darkest hours, when the end seemed so far away and it was tempting to abandon the fight. This dissertation and my life are the richer because of her. She is the light of my life.

Table of Contents

List of Tables

List of Figures

Chapter 1

The Problem of Locality

Modern high-performance computers are organized around an extremely fast processor and a slower memory system. To present the illusion of a faster memory system to the processor, it is common to organize the memory in the system as a deep hierarchy(Figure 1.1). The memory hierarchy is organized as a series of levels — faster levels contain less memory than slower levels. The overall goal is to give the processor the impression of having a memory as large as the largest level of the memory hierarchy, with the access time of the fastest level [27].

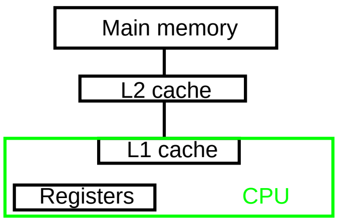


Figure 1.1: Modern machines with deep memory hierarchies

Table 1.1: Typical memory access times in processor cycles

	Registers	L1(on chip)	L2	Main memory
SGI Octane (R10K)	1	2-4	10	70
PC (Pentium II)	1	1-2	≈ 30	≈ 60
Single SP-2 node (P2SC)	1	–	3-4	≈ 60

There are three points worth noting about the data in Table 1.1.

- All machines have memory hierarchies that are three or four level deep.
- The data access time from a given level of the memory hierarchy to the next slower level increase by as much as a factor of 20.
- All levels of the memory hierarchy are not equally crucial - some have potentially more impact that others.

Given the order-of-magnitude increase in access time from one level of the memory hierarchy to the next, it is imperative to move the data efficiently through the memory hierarchy. For example, consider the program fragment in Figure 1.2. `Sum` is a scalar that is typically register allocated. On a modern processor this code would be compiled to a load instruction to load the value of `a(i)` from memory and an `add` instruction to add the loaded value to the scalar `sum`. The `add` instruction typically takes a single cycle on a modern microprocessor, whereas the latency of the load instruction depends on which level of the memory hierarchy contains `a(i)`. For a computer with two levels of cache where the Level 1 cache has a latency of 1

```
do i = 1, n
  sum = sum + a(i)
end
```

Miss Ratio (%)	Execution Time (Cycles)	Idle Time (%)	Slow-down (%)
0	n	0	0
5	$1.45n$	26	45
10	$1.9n$	47	90
20	$2.8n$	64	180
50	$5.5n$	81	450
100	$10n$	90	900

Figure 1.2: A simple test program and its execution time for different miss ratios

cycle, and the Level 2 cache has a latency of 10 cycles, the table in Figure 1.2 shows the program execution time in processor cycles for various hit ratios in the Level 1 cache - for as small a miss ratio as 5%, there is a slowdown in execution time by 45%.

Present compiler technology is based on the linear loop transformation framework for perfectly nested loops. A perfectly nested loop is a loop nest in which all the assignment statements are contained inside all the loops, and includes simple programs such as matrix matrix product and stencil computations. However because of their very simple structure, the class of perfectly nested programs can only represent either very simple programs or innermost kernels in complex programs. Therefore, a systematic framework is desirable for dealing with a more general class of programs. While several solutions have been suggested that convert non-perfectly nested loops into perfectly nested loops by inserting conditionals, the manner in which the conditionals are inserted has an unpredictable effect on subsequent transformations that can be applied. In other words, there is no systematic way in which a non-perfectly nested loop can be converted into a perfectly nested loop that can be transformed by existing technology.

In this context, the contributions of this dissertation are:

- Presenting a framework for enhancing locality by solving the problem in the domain of the data, rather than in the iteration space. This is the basis of the **data-centric** approach to locality. The key point is that rather than driving the locality enhancement process by concentrating on transformations of the loops, an elegant and comprehensive solution can be obtained by driving the process by focusing on the data instead.

- Showing that the data-centric approach can be used not just to enhance locality, but also to derive **parallelism**. In fact, for the class of semi-structured applications, it is shown that using the data-centric approach, parallel code competitive with hand-written library code can be derived automatically.

In summary, the contribution of this thesis is a new approach called the *data-centric approach* to locality enhancement, with potential generalizations to automatic parallelization.

Chapter 2

Approaches to Exploiting Locality

2.1 Library Approaches

The scientific computing community has a rich selection of problems which perform computation on large volumes of data. Consequently, to achieve good performance for these applications, it is necessary to be very careful about exploiting locality in the memory hierarchy. To allow for portable development of high-performance software, researchers have assembled a short list of core operations on which applications can be layered. These core operations have different computational and memory behavior and hence have different levels of performance. Since applications are usually layered on top of these core routines, it is critical to understand these basic operations to reason about the performance of applications. Six core operations are especially common [24]:

- **Dot product**: Given two column vectors x and y, computes the inner product $x^T y$.
- **Saxpy**: Given a scalar α and column vectors x and y, computes the column vector z equal to $\alpha * x + y$ (one can think of saxpy as shorthand notation for *scalar α times vector x plus y*).
- **Matrix Vector Product**: Given an $m \times n$ matrix A and a column vector x, computes $y = A * x$.
- **Outer Product Update**: Given an $m \times n$ matrix A and column vectors x and y, computes $A = A - xy^T$.
- **Gaxpy**: Given an $m \times n$ matrix A, and vectors x, y, computes $z = y + A * x$. It is easy to see that a gaxpy operation represents a series of saxpy operations, where each column of A is scaled by a single entry of x.
- **Matrix Multiplication**: Given an $m \times n$ matrix A, an $n \times p$ matrix B and an $m \times p$ matrix C, computes $C = C + A * B$.

Figure 2.1 shows potential conceptual implementations of each of these core routines.

Three important properties of each of these core routines need to be studied to understand the performance - (i) amount of data touched, (ii) the number of floating point operations, and (iii) the average amount of data reuse (which is the ratio of the two previous quantities). Table 2.1 summarizes this information for each of the core operations.

The information in Table 2.1 provides a guide to the potential performance of each of these core operations. A key characteristic of each operation is its associated "level", which can be defined as the exponent of n which represents the amount of

(a) Inner (dot) Product

```
sum = 0
do i = 1, n
  sum += x(i) * y(i)
```

(b) Scalar * x + y (Saxpy)

```
do i = 1, n
  z(i) += α * x(i) + y(i)
```

(c) Matrix Vector Product

```
do i = 1, m
  do j = 1, n
    y(i) += A(i,j) * x(j)
```

(d) Outer product update

```
do i = 1, n
  do j = 1, m
    A(i,j) += x(i) * y(j)
```

(e) Matrix * x + y (Gaxpy)

```
do i = 1, m
  do j = 1, n
    z(i) += y(i) + A(i,j) * x(j)
```

(f) Matrix Multiplication

```
do i = 1, m
  do j = 1, n
    do k = 1, p
      C(i,j) += A(i,k) * B(k,j)
```

Figure 2.1: Common core operations in scientific computing

Table 2.1: Behavior of six core operations in scientific computing

Core operation	Data Used	Computation	Average reuse	"Level"
$x^T y$	$O(n)$	$O(n)$	x, y: $O(1)$	1
$z = \alpha * x + y$	$O(n)$	$O(n)$	x, y, z: $O(1)$	1
$A * x$	$O(n^2)$	$O(n^2)$	x: $O(n)$, A: $O(1)$	2
$A = A + xy^T$	$O(n^2)$	$O(n^2)$	x, y: $O(n)$, A: $O(1)$	2
$z = x + A * y$	$O(n^2)$	$O(n^2)$	x, y: $O(n)$, A: $O(1)$	2
$C = C + A * B$	$O(n^2)$	$O(n^3)$	A, B, C: $O(n)$	3

floating point operations in the computation. For example, dot product and saxpy perform $O(n)$ computation and are level-1 computations. Matrix vector product, outer product update and gaxpy perform $O(n^2)$ operations and are level-2 computations. Finally, matrix multiplication performs $O(n^3)$ operations and is a level-3 computation.

The **level** of a computation provides a quick and succinct insight into the performance of the computation - *higher the level, better the performance*. For example, a level-1 operation such as inner product performs a total of $2 * n$ floating point operations on $2 * n$ data elements. There is no reuse for either of the two vectors involved in the computation. A level-2 operations such as matrix vector product, on the other hand performs $2 * n^2$ operations on $n^2 + 2 * n$ data elements. While there is no reuse for the matrix involved in the computation, each of the vectors exhibits $O(n)$ amount of reuse. In addition, the amount of data touched for a given number of floating point operations is half the data touched in the case of a level-1 computation performing the same number of operations for large values of n. Finally, matrix multiplication performs $2 * n^3$ operations on $3 * n^2$ data elements, while exhibiting $O(n)$ amount of reuse for each of the three matrices involved in the computation. Conversely, the amount of data touched in matrix multiplication for a given number of floating point operations is significantly smaller than that touched by a level-1 or level-2 operation for the same number of operations.

Given this computational and memory behavior of the core operations, one would expect a level-3 computation to outperform a level-2 computation, and a level-2 computation to outperform a level-1 computation on a system where the

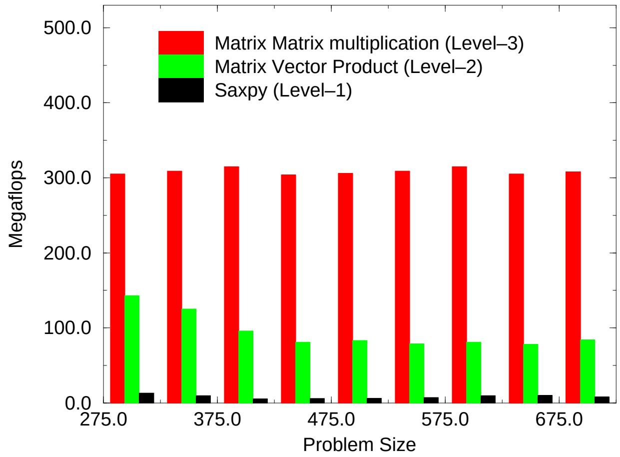


Figure 2.2: Performance of computations from Levels 1, 2 & 3

memory speed is significantly slower than the processor speed. This is in fact the case on modern high-performance computers. Figure 2.2 shows the performance of three computations (one each from levels 1, 2 & 3) on an Octane workstation from Silicon Graphics, equipped with a 195 MHz R10000 processor, an 1MB 2-way set associative secondary cache with a line size of 128 bytes and a 32-KByte 2-way set associative primary data cache with a line size of 32 bytes. The R10000 can perform 1 load/store and two floating point operations per cycle, giving it a peak performance of 390 MFlops. It has 32 logical floating point registers, which can be mapped to any of 64 physical registers using register renaming. As shown in Figure 2.2, matrix multiplication achieves approximately 300 Mflops (about 75% of peak performance). It significantly outperforms the level-2 matrix vector product (running at around 80 Mflops), which in turn significantly outperforms the level-1 saxpy operation which runs at about 10 Mflops.

As a result, the numerical analysis community has used a layered approach to the problem of writing portable software with good data reuse. Having defined a core set of useful operations, the general idea is to (a) implement carefully hand-tuned implementations of these algorithms on all platforms, and (b) use those operations, wherever possible, in writing programs for applications. From Amdahl's law, it follows that if most of the computational work of a program is done in the core operations, the program will perform well on a machine with a memory hierarchy. While the implementations of these operations are not portable, the rest of the software is machine-independent.

The performance data in Figure 2.2 leads to another critical design criterion for developing applications in the domain of dense linear algebra - the wide variation in performance from operations that fall into different levels. For example, even though a gaxpy operation can be conceptually thought of as a series of saxpy operations, a gaxpy is a level-2 operation, whereas a saxpy is a level-1 operation. This suggests that even though the computation performed may be equivalent, a single gaxpy operation will achieve higher performance than the same computation written as a series of saxpy operations. Similarly, while a matrix multiplication can be thought of conceptually as a series of gaxpy operations, a single matrix multiplication routine can perform significantly better than the same computation implemented as a series of gaxpy operations.

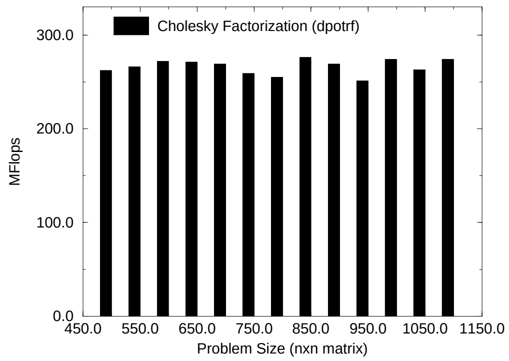
From this discussion, it should be clear that in the context of dense numerical linear algebra, matrix multiplication should be exploited wherever possible. The standard algorithm for multiplying two $n \times n$ matrices performs n^3 operations on n^2 data, so it has excellent data reuse. Most vendors provide hand-tuned efficient implementations of matrix matrix multiplication and other core routines conforming to a standard interface called the Basic Linear Algebra Subroutine package (also known as the BLAS routines [35]).

There is a final issue regarding performance: while matrix multiplication offers excellent reuse for each of the three matrices involved, the naive version in Figure 2.1(f) performs poorly in practice. This is because the entire potential reuse in all the three arrays is not exploited. For every iteration of the outermost loop, the matrix `B` needs to be read in its entirety. If the matrices `A` and `C` are large enough to fill the cache, then none of the reuse for `B` is exploited. Researchers have discovered that this problem can be addressed by reassociating the computation in matrix multiplication using a *block-matrix* algorithm. A *block* in a matrix is essentially a submatrix. In a block-matrix version of matrix multiplication, one computes the contribution to blocks of `C` by taking blocks of `A` and `B` one at a time, instead of computing the final value of the matrix `C` element-by-element. More details can be found in any standard text book on numerical analysis such as [24]. Detailed information regarding the implementation of BLAS on a modern high-performance computer such as the IBM RS6000 system can be found in [1].

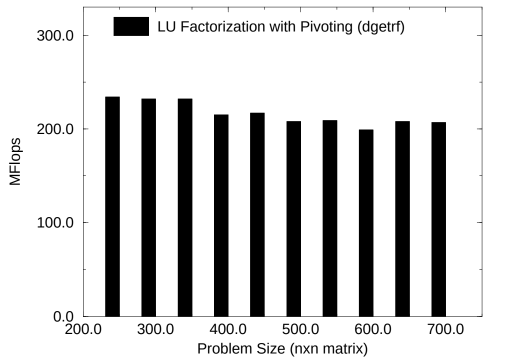
While the BLAS routines provide efficient implementations of several core operations from scientific computing, there are several other algorithms that applications use often. These are:

- **Cholesky Factorization**: Cholesky factorization is used to solve the system of equations `Ax = b`, where `A` is a symmetric positive-definite matrix, by factorizing A into the product $\mathtt{LL}^{\mathrm{T}}$, where `L` is lower-triangular, and solving the two resulting triangular systems. To save space, the lower triangular part of `A` is overwritten with the factor `L`.
- **LU Factorization with partial pivoting**: LU factorization is used to solve general systems of equations of the form `Ax = b` by factoring `A` into the product `LU` where `L` is a lower-triangular matrix and `U` is upper-triangular. Partial pivoting is used to increase the numerical stability of the procedure.
- **QR Factorization with Householder reflections**: QR factorization performs orthogonal factorization of a matrix `A` into the product `QR` where `Q` is an orthonormal matrix and `R` is upper triangular. It is a key kernel in eigenvalue calculations.

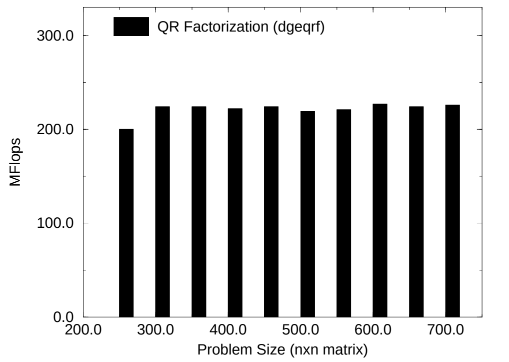
Just as matrix multiplication needs to be reorganized using block matrix methods, to exploit performance in the factorization routines, the underlying computation must be reorganized to perform operations on block matrices at a time. However, while this reorganization is trivial in the case of matrix multiplication, it is significantly non-trivial for the factorization routines. The numerical analysis community has invested considerable effort in developing these block-matrix routines by hand. In many cases, the block-matrix algorithms require algebraic restructuring of the underlying computation, such as utilizing the fact that matrix multiplication is associative. By operating on entire submatrices at a time, the block-matrix algorithms are rich in matrix multiplication operations and can utilize the BLAS library



(a) Cholesky Factorization



(b) LU Factorization with Partial Pivoting



(c) QR Factorization using Householder Reflections

Figure 2.3: Performance of Cholesky, LU & QR Factorization codes from LAPACK

to achieve high performance.

The well-known LAPACK library contains block matrix algorithms implemented on top of the BLAS routines, and is written for good data reuse on a machine with a two-level memory hierarchy [3]. Figure 2.3 shows the performance of the three factorization routines from the LAPACK library on an SGI Octane workstation (as before, equipped with a 195 MHz R10000, a 1MB level 2 cache and a 32KB level 1 cache). It is worth noting that these operations do not quite achieve the same level of performance as matrix multiplication on the same platform, because in addition to level-3 operations, they also perform level-2 and level-1 operations, and Amdahl's law comes into place.

The LAPACK library has been successful in practice. However, it requires a set of machine-specific, hand-coded BLAS routines to run well. In addition, it requires considerable human effort to derive blocked versions of the factorization routines. Since it is not a general-purpose tool, it cannot be used outside the realm of the dense numerical linear algebra. It is also specifically written for a two-level memory hierarchy, and it must be re-implemented for machines with deeper memory hierarchies. Therefore, automatic program restructuring tools that promote data reuse through program transformations provide an attractive alternative.

2.2 Approaches in Production Compilers

The restructuring compiler community has devoted much attention to the development of such technology. The most important contribution is Mike Wolfe's *iteration* *space tiling* [48], preceded by *linear loop transformations* if necessary [7, 31, 45]. This approach is restricted to perfectly nested loops, and some extensions to simple non-perfectly nested loops have also been proposed. This technology is currently very mature and several production compilers, including those from Silicon Graphics, Hewlett Packard and International Business Machines routinely use this technology to improve program performance.

A loop in a loop nest is said to *carry reuse* if the same data is touched by multiple iterations of that loop for fixed outer loop iterations. For example, loop `k` in Figure 2.4(b) carries reuse because for fixed `i` and `j`, all iterations of the loop touch `C(i,j)`; similarly, loop `i` carries reuse because successive iterations of the `i` loop touch `B(k,j)`. Loops that carry reuse are moved as far inside the loop nest as possible by using linear loop transformations; if two or more inner loops carry reuse and they are *fully permutable*, these loops are tiled [48]. Intuitively, tiling improves performance by interleaving iterations of the tiled loops, which exploits data reuse in all those loops rather than in just the innermost one. It is easy to verify that all the three loops in the matrix multiplication code carry reuse and are fully permutable. Tiling all three loops produces the code shown in Figure 2.4(c) (for 25×25 tiles). The three outer loops enumerate the iteration space tiles, while the three inner loops enumerate the iteration space points within a tile. In this case, iteration space tiling produces the same code as the equivalent block matrix code [24].

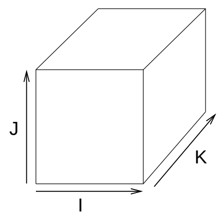
Matrix multiplication represents a simple case of tiling – in general, tiling need only be preceded by loop interchange. Since there are three nested loops in matrix

(a) Program model for perfectly nested loops

```
do i_1 = l_1, u_1
  do i_2 = l_2, u_2
    do i_n = l_n, u_n
      s1(i_1, i_2, ..., i_n)
      s2(i_1, i_2, ..., i_n)
      ...
      sn(i_1, i_2, ..., i_n)
```

(b) Matrix Multiplication - Untiled Version

```
do i = 1, n
  do j = 1, n
    do k = 1, n
      C(i,j) += A(i,k) * B(k,j)
```



(c) Matrix Multiplication - Tiled Version (25x25x25 tiles)

```
do t1 = 1, ⌈(n/25)⌉
  do t2 = 1, ⌈(n/25)⌉
    do t3 = 1, ⌈(n/25)⌉
      do it = (t1-1)*25 +1, min(t1*25,n)
        do jt = (t2-1)*25 +1, min(t2*25,n)
          do kt = (t3-1)*25 +1, min(t3*25,n)
            C(it,jt) += A(it,kt)*B(kt,jt)
```

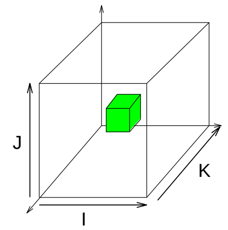


Figure 2.4: Matrix Multiplication: ijk version

multiplication, and all possible permutations of these are legal, determining the correct permutation sequence to precede tiling is a simple problem. However, for more general perfectly nested loops, it may be necessary to precede tiling with any of the general linear loop transformations including loop skewing, loop reversal and loop scaling in addition to loop permutation. There are two equivalent approaches to dealing with this situation – (i) All loop transformations act on the loop nest before

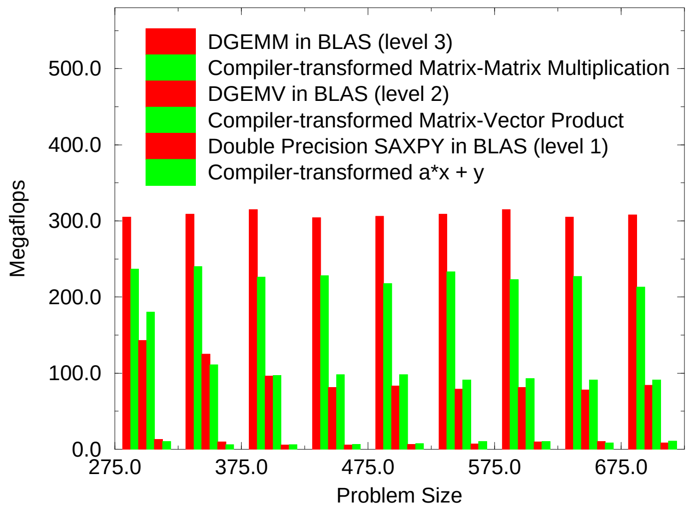


Figure 2.5: Tiling vs. Handwritten BLAS code for Level-1, 2 & 3 Computations

tiling is applied, and the resulting iteration space is tiled using cutting planes parallel to the axes. In the case, tiling is syntactically performed by a combination of strip-mine and interchange transformations. (ii) Alternatively, tiling can be performed on the initial iteration space by choosing arbitrarily oriented equidistant sets of cutting planes. In the second approach, the effect of loop transformations preceding tiling is emulated by choosing arbitrarily oriented cutting planes.

To fully automate tiling, cutting plane orientations and their separations must be determined automatically. Three important criteria must be kept in mind while determining tile sizes – (i) It is desirable for the data touched in a tile to fit in the level of cache under consideration so as to minimize capacity missed. (ii) It is also desirable to choose tile sizes and orientations so as to minimize interference between the various data arrays accessed in a given tile, so as to minimize conflict misses. (iii) Finally, to maximize reuse of data within a tile, it is desirable to optimize some relationship between the surface and volume of a tile – the surface of a tile in conjunction with dependences in the program yields information about how much data is shared between tiles, while the volume yields information about the amount of work performed within a tile. Clearly, it is desirable to minimize data sharing between tiles while maximizing the amount of work performed within a tile.

Much research has been devoted to this effort. Dongarra and Schreiber [19] determine tile sizes and orientations such that the amount of data touched by the tile is bounded by the size of the cache, while minimizing the surface to volume ratio of the tile. The solution is developed by modeling the problem as a constrained minimization problem. Darte [8] suggests that the surface to volume ratio is not the appropriate metric to be minimized and present alternative metrics one might choose to optimize. Neither of these two approaches addresses the question of conflict misses. Lam, Rothberg and Wolf present strategies to determine square tiles while minimizing capacity and conflict misses [30]. This is generalized to determining rectangular tiles while minimizing capacity and conflict misses by Coleman and McKinley [17]. Wolf, Chen and Maydan [46] discuss how tile sizes can be determined effectively and efficiently in the setting of production compilers.

Tiling has been around in the compiler community for almost a decade and is mature technology today. In fact, commercial compilers such as those from Silicon Graphics(SGI) and International Business Machines(IBM) apply extensive tiling and other enabling loop transformations [46, 42] on simple program fragments such as those from Figure 2.1. Figure 2.5 compares the effectiveness of tiling and enabling loop transformations on matrix multiplication, matrix vector product and inner product against that of handwritten code for the same computations. It can be seen that tiling performs very creditably - it achieves 80% of the performance of the handwritten code for matrix multiplication and actually outperforms the handwritten code for matrix vector product and inner product computation. This raises a very interesting question - if tiling can be used to automatically achieve performance competitive with handwritten BLAS library, can it achieve similar results for computations in LAPACK?

2.2.1 Effectiveness of Tiling on LAPACK Codes

As already mentioned, three codes - *Cholesky Factorization*, *LU Factorization with pivoting* and *QR Factorization* are the three basic building blocks in LAPACK. Unfortunately, the implementations of all these factorization routines correspond to non-perfectly nested loop programs. Hence, tiling as described in Section 2.2 cannot be directly applied to these programs. In this context, the most practical generalization of tiling to non-perfectly nested loops has been for the class of *singly nested loops(SNLs)*. This is the class of programs represented in Figure 2.6. The key idea is that tiling can be applied to imperfectly nested loops if these loops are converted to perfectly nested loops through the use of *code sinking* [49]. Conceptually, code sinking moves all statements into the innermost loop, inserting appropriate guards to ensure that these statements are executed the right number of times. This is shown in Figure 2.7 for a simple singly nested loop. Code sinking for the class of singly nested loops is well understood and dealt with in [47] in detail. Code sinking followed by tiling for SNLs has been integrated into commercial production compilers such as at Silicon Graphics (SGI) [46]. This section evaluates the effectiveness of a state-of-the-art implementation of tiling in the presence of all other optimizations in the production compiler from Silicon Graphics for *Cholesky factorization*, *LU Factorization with pivoting* and *QR factorization*. As before, all experiments have been conducted on an unloaded Octane workstation with a 1MB level 2 cache and a 32 KB level 1 data cache.[1].

2.2.2 Cholesky factorization

Cholesky factorization, like matrix multiplication, has three nested loops although these loops are imperfectly nested. All six permutations of these loops are legal, but some of these permutations are not SNL's. These permutations are shown in Figure 2.8 through Figure 2.12. As discussed in Section 2.1, LAPACK library code for Cholesky Factorization obtains $\approx$ 260 MFlops.

The most commonly described version is the so-called `kij` version, and it is shown in Figure 2.8(a). This version processes the columns of the matrix in left to right order as follows: the square root of the diagonal element of the current column is computed, the portion of this column below the diagonal is scaled with this value and the outer-product of this portion of the column with its transpose is used to update the lower triangular portion of the matrix to the right of the current column.

[1]The flags used were -O3 -n32 -mips4, and tiling was turned on or off using the -LNO:blocking= flag

```
do i_1 = l_1, u_1
  s1a(i_1)
  do i_2 = l_2, u_2
    s2a(i_1,i_2)
    ...
    do i_n = l_n, u_n
      sn(i_1, i_2, ..., i_n)
    ...
    s2b(i_1,i_2)
  s1b(i_1)
```

(a) Program model for singly nested loops

```
do i = 1, n
  y(i) = 0
  do j = 1, n
    y(i) += A(i,j) * x(j)
  y(i) = c * y(i)
```

```
do i = 1, n
  y(i) = 0
  do j = 1, n
    y(i) += A(i,j) * x(j)
  do j = 1, n
    y(i) += B(i,j) * x(j)
```

(b) Singly nested loop (left) and Non-singly nested loop (right)

Figure 2.6: Singly Nested Loops (SNLs) - (a) Program Model, (b) Examples

```
do i = 1, n
  S1(i)
  do j = 1, n
    S2(i,j)
  S3(i)
```

(a)

```
do i = 1, n
  do j = 1, n
    if (j == 1) S1(i)
    S2(i,j)
    if (j == n) S3(i)
```

(b)

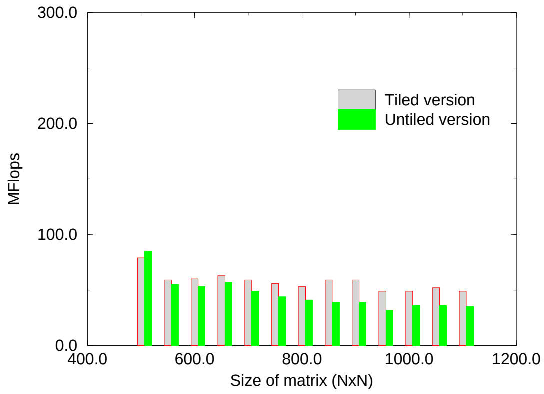
Figure 2.7: Code Sinking for an SNL – (a) Before, (b) After

These are known as the *square root*, *scale* and *update* steps respectively. For obvious reasons, this version of Cholesky factorization is also known as *right-looking column Cholesky factorization*. Notice that this code is not an SNL. The SGI compiler does not fuse the scale and update loops. Most of the computation is performed by the update loop, but each iteration of the loop nest writes to a different location, so there is little temporal locality. However, tiling has a small benefit because it permits spatial locality to be exploited for all the references in the loop (the bar named "Untiled version" is the performance of the code produced by the compiler with all optimizations other than tiling turned on). Jamming the scale and update loops in Figure 2.8(a) produces the SNL shown in Figure 2.8(b) which can be tiled by the SGI compiler, doubling performance.

The `kij` versions update the right lower-triangular portion of the matrix row-by-row. Permuting the two update loops gives the *kji* version shown in Figure 2.9 that performs this update column-by-column. This version is not an SNL, so tiling is not effective. Fusing the scale loop with the outer update loop is illegal. The only way to get an SNL is to interchange the two update loops and then fuse the new outer update loop with the scale loop, generating the code of Figure 2.8(b) in effect, but this is too complicated for the compiler to reason about. The performance of the baseline code is modestly better than that of the baseline `kij` versions because of better spatial locality in the update loops.

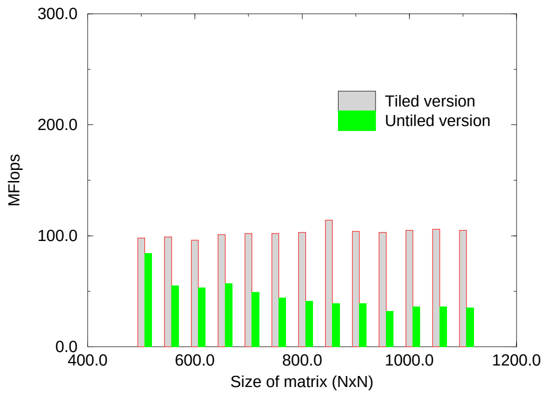
Right-looking Cholesky factorization performs updates *eagerly* in the sense that the columns to the right of the current column are updated as soon as that column is computed. An alternative is to perform the updates *lazily*, which means that a

```
do k = 1, NMAX
  A(k,k) = dsqrt (A(k,k))
  do i = k+1, NMAX
    A(i,k) = A(i,k) / A(k,k)
  do i = k+1, NMAX
    do j = k+1, i
      A(i,j) -= A(i,k) * A(j,k)
```



(a) Cholesky factorization: Version kij (distributed)

```
do k = 1, NMAX
  A(k,k) = dsqrt (A(k,k))
  do i = k+1, NMAX
    A(i,k) = A(i,k) / A(k,k)
    do j = k+1, i
      A(i,j) -= A(i,k) * A(j,k)
```



(b) Cholesky factorization: Version kij (fused)

Figure 2.8: Cholesky Factorization: kij version

```
do k = 1, NMAX
  A(k,k) = dsqrt (A(k,k))
  do i = k+1, NMAX
    A(i,k) = A(i,k) / A(k,k)
  do j = k+1, NMAX
    do i = j, NMAX
      A(i,j) -= A(i,k) * A(j,k)
```

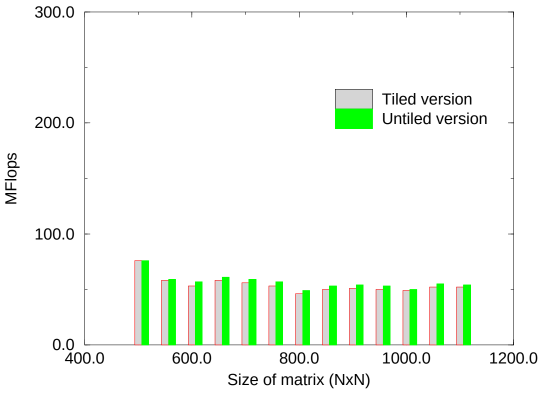


Figure 2.9: Cholesky Factorization: kji version

column is updated only when it becomes current. This leads to the *left-looking column Cholesky factorization* code (also called the `jik` version) shown in Figure 2.10 which applies updates from all columns to the left of the current column before performing the square root and scaling steps. The loop nest is not an SNL, but the computational work in the update loops is essentially a matrix-vector product which is performed efficiently by the SGI compiler by accumulating the updates to `A(i,j)` in a register.

In the `jik` version, all the updates to an element of the current column are performed before succeeding elements are updated. Permuting the `i` and `k` loops gives the `jki` version shown in Figure 2.11. The SGI compiler interchanges the update loops back to the `jik` version, so the performance of the baseline and tiled versions is identical to the performance of the `jik` versions.

Finally, there are two versions of Cholesky factorization called the `ijk` and `ikj` versions that process the matrix by row rather than by column. The `ijk` version

```
do j = 1, NMAX
  do i = j, NMAX
    do k = 1, j-1
      A(i,j) -= A(i,k) * A(j,k)
   A(j,j) = dsqrt (A(j,j))
   do i = j+1, NMAX
     A(i,j) = A(i,j) / A(j,j)
```

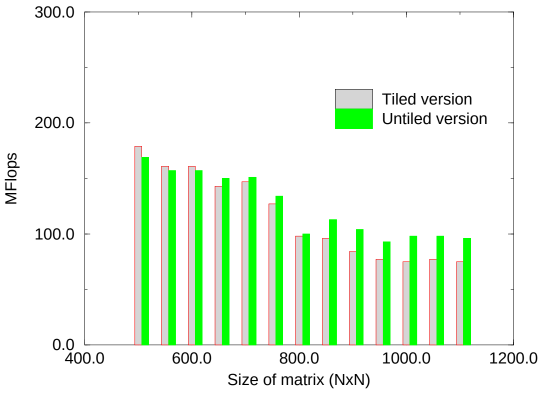


Figure 2.10: Cholesky Factorization: Version jik

performs inner-products, so it is also known as *dot Cholesky* while the `ikj` version in contrast is rich in daxpy operations. The compiler is unable to reorganize the *dot product* computation in the `ijk` version in any meaningful way, resulting in the extremely poor performance.

2.2.3 LU factorization

The computation in LU factorization with no pivoting is similar to that of Cholesky factorization with two caveats: LU factorization is applied to matrices not guaranteed to be symmetric positive-definite, and hence it is not guaranteed that the diagonal elements are always positive at any given stage. Hence, the square root step in Cholesky factorization is absent. Secondly, since the matrix is not symmetric, the outer product update is applied to the entire submatrix on the right hand side of the current column, and not just the lower triangular portion. Because of these reasons, LU factorization without pivoting is subsumed by the discussion

```
do j = 1, NMAX
  do k = 1, j-1
    do i = j, NMAX
      A(i,j) -= A(i,k) * A(j,k)
  A(j,j) = dsqrt (A(j,j))
  do i = j+1, NMAX
    A(i,j) = A(i,j) / A(j,j)
```

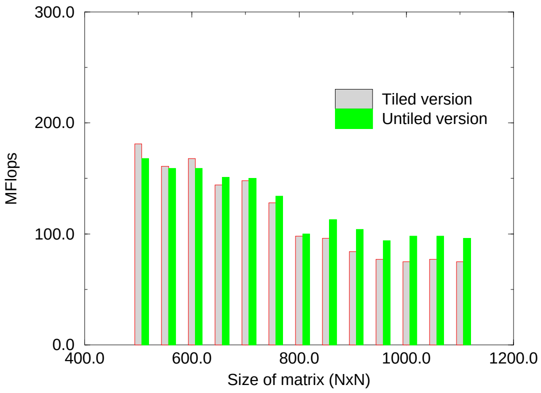


Figure 2.11: Cholesky Factorization: Version jki

regarding Cholesky factorization.

LU factorization without pivoting is not numerically stable. To remedy this, *partial pivoting* is introduced. Partial pivoting works as follows. As in the case of Cholesky factorization, when a column is visited, the current diagonal element (also called the *pivot position*) is used to divide the entries of the current column in LU factorization. It turns out that small values are undesirable for the pivot element and lead to numerical instability. The purpose of the partial pivoting operation is to find the entry in the current column with the largest magnitude and apply row permutations to bring this entry to the pivot position. While it still remains an open problem to prove that partial pivoting does indeed lead to numerical stability, it behaves well enough in practice that this factorization is a key computational kernel. As discussed in Section 2.1, LAPACK library code for LU Factorization with Partial Pivoting obtains approximately 225 MFlops.

LU factorization with pivoting, like Cholesky factorization and matrix multiply,

```
do i = 1, NMAX
  do j = 1, i-1
    do k = 1, j-1
      A(i,j) -= A(i,k) * A(j,k)
    A(i,j) = A(i,j)/A(j,j)
  do k = 1, i-1
      A(i,i) -= A(i,k) * A(i,k)
  A(i,i) = dsqrt (A(i,i))
```

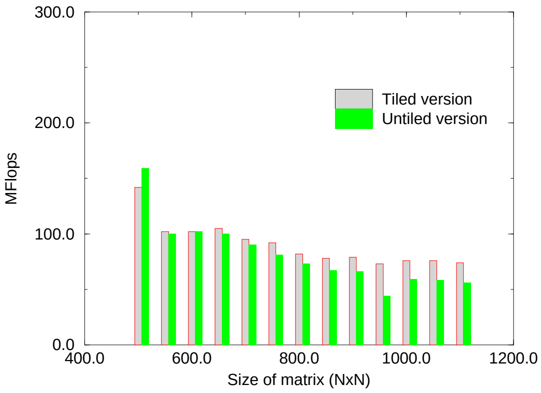


Figure 2.12: Cholesky Factorization: Version ijk

has three nested loops. Because of dependences imposed by row permutation, the outermost `k` loop cannot be permuted with either of the update loops (`i` and `j`) loops. However, the update loops themselves can be interchanged, giving rise to two different versions of LU factorization with pivoting.

Figure 2.15 and Figure 2.16 show the benefit from tiling for the `kij` and `kji` versions of LU factorization with pivoting. The entire loop nest is not an SNL, and therefore cannot be tiled. However, the update loop nest can be tiled, and this has some benefit because it permits spatial locality to be exploited. It should also be noted that both the tiled and untiled instances of the `kij` perform somewhat better than the `kji` version because of improved spatial locality in the update loops.

2.2.4 QR factorization

QR factorization computes `QR = A`, where `Q` is an orthonormal matrix and `R` is upper triangular. The Householder variant of QR factorization proceeds through the

```
do i = 1, NMAX
  do k = 1, i-1
    A(i,k) = A(i,k) / A(k,k)
    do j = k+1, i
      A(i,j) -= A(i,k) * A(j,k)
  A(i,i) = dsqrt (A(i,i))
```

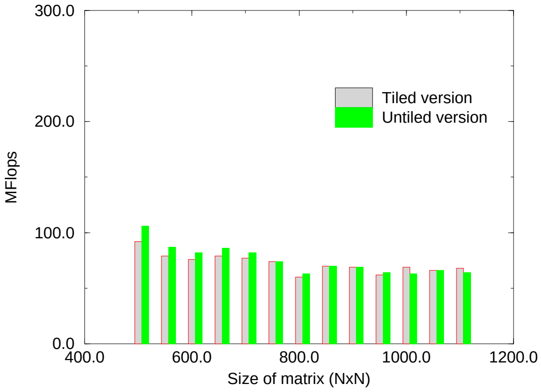


Figure 2.13: Cholesky Factorization: Version ikj

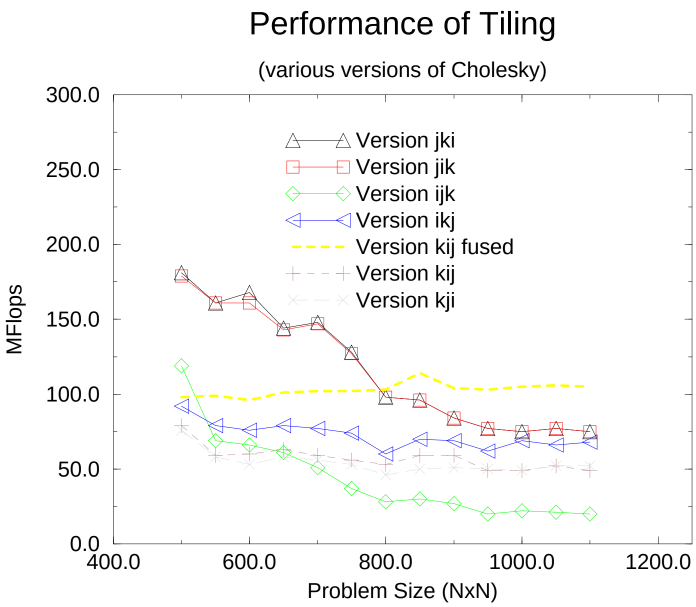


Figure 2.14: Effectiveness of Tiling on different versions of Cholesky

```
do k = 1, n
  temp = 0.0d0
  m = k
  //find pivot row
  do i = k, n
    d = A(i,k)
    if (ABS (d) .gt. temp)
      temp = abs(d)
      m = i
  if (m .ne. k)
    ipvt(k) = m
    //row permutation
    do j = k, n
      temp = A(k,j)
      A(k,j) = A(ipvt(k),j)
      A(ipvt(k),j) = temp
  //scale loop
  do i = k+1, n
    A(i,k) = A(i,k) / A(k,k)
  //update loops
  do i = k+1, n
    do j = k+1, n
      A(i,j) -= A(i,k) * A(k,j)
```

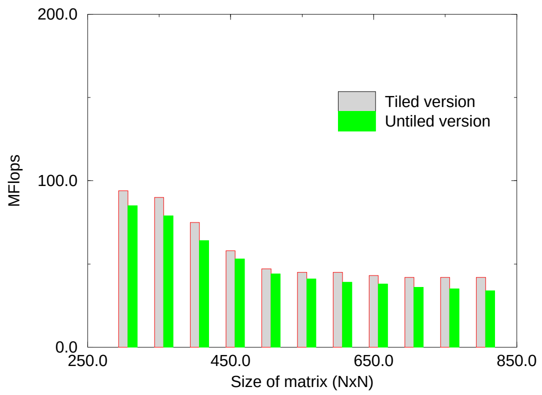


Figure 2.15: LU Factorization with Partial Pivoting - kij version

matrix A column by column. For each column, a Householder vector is determined such that upon the application of a Householder reflection matrix, the portion of the current column below the diagonal contains only zeros. For a vector x, if e_1 represents the unit vector with a 1 in the first entry, and zeros in all other entries, $v = (x - \|x\|_2 * e_1)/\|(x - \|x\|_2 * e_1)\|$ represents a unit-length Householder vector such that on applying a Householder reflection to x, all entries except the first are zeroed out. Once a Householder vector v has been determined for the current column, a Householder reflection can be applied to the rest of the matrix. Conceptually, a

```
do k = 1, n
  temp = 0.0d0
  m = k
  //find pivot row
  do i = k, n
    d = A(i,k)
    if (ABS (d) .gt. temp)
      temp = abs(d)
      m = i
  if (m .ne. k)
    ipvt(k) = m
    //row permutation
    do j = k, n
      temp = A(k,j)
      A(k,j) = A(ipvt(k),j)
      A(ipvt(k),j) = temp
  //scale loop
  do i = k+1, n
    A(i,k) = A(i,k) / A(k,k)
  //update loops
  do j = k+1, n
    do i = k+1, n
      A(i,j) -= A(i,k) * A(k,j)
```

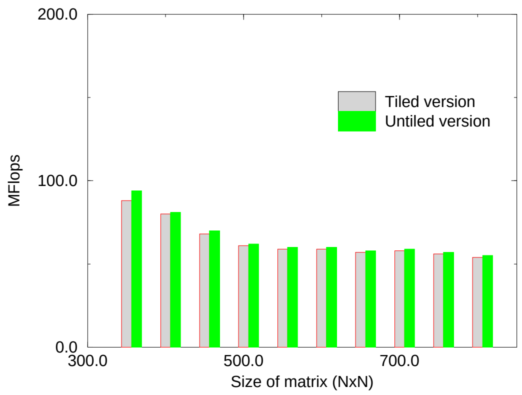


Figure 2.16: LU Factorization with Partial Pivoting - kji version

Householder reflection can be thought of as multiplying the rest of the matrix by $(I - 2vv^T)$, which would take $O(n^3)$ operations for v of length n. However, because of the special structure of the Householder matrix, this reflection can actually be accomplished using only $O(n^2)$ operations by using a two-step process. The key observation is that for a matrix B, $(I - 2vv^T) * B = B - 2v * v^T * B = B - 2v * w$, where $w = v^T * B$. Thus the first step computes w using a matrix vector computation and the second step updates the rest of the matrix using an outer product update computation. As discussed in Section 2.1, LAPACK library code

```
do k = 1, n
  norm = 0
  do i = k, n
    norm = norm + A(i,k) * A(i,k)
  norm2 = dsqrt (norm)
  asqr = A(k,k) * A(k,k)
  A(k,k) =
    dsqrt(norm-asqr+((A(k,k)-norm2)^2))
  //Householder vector computation
  do i = k+1, n
    A(i,k) = A(i,k) / A(k,k)
  //Reflection - (matrix vector product)
  do i = k+1, n
    w(i) = 0
    do j = i, n
      w(i) += A(j,i) * A(j,k)
  //Reflection - (outer product update)
  do j = k+1, n
    do i = k+1, n
      A(i,j) = A(i,j) - 2 * A(i,k) * w(j)
```

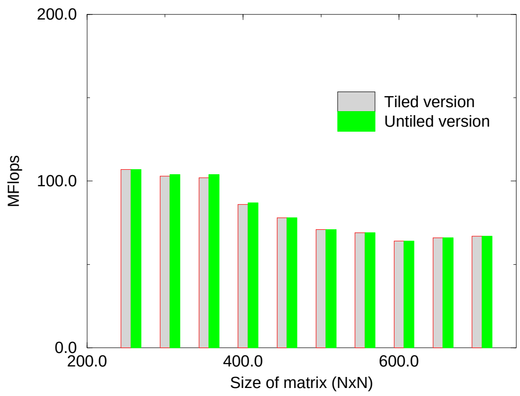


Figure 2.17: QR Factorization using Householder reflections

for QR Factorization obtains approximately 225 MFlops.

The entire computation represents a non-perfectly nested loop, which cannot be tiled. The matrix vector product and outer product update operations as written in the initial code already exploit complete spatial locality, and no additional benefit is obtained from tiling, which is evident from Figure 2.17.

2.3 Research Compiler Approaches

While Section 2.2 discusses existing compiler technology mature enough to be integrated into production compilers, it is by no means the only technology currently

```
do i = 1, m
  do j1 = 1, n
    do k1 = 1, p
      a(i,j1,k1) = ...
  do j2 = 1, n
    do k2 = 1, p
      ... = a(i,j2,k2)
```

(a)

```
do i = 1, m
  do j = 1, n
    do k = 1, p
      a(i,j,k) = ...
      ... = a(i,j,k)
```

(b)

Figure 2.18: Loop Fusion - (a) Before & (b) After Fusion

```
do i = 1, m
  do j1 = 1, n
    do k1 = 1, p
      a(i,j1,k1) = ...
  do j2 = 1, n
    do k2 = 1, p
      ... = a(i,j2,k2)
```

(a)

```
do i = 1, m
  do j1 = 1, n
    do k1 = 1, p
      a(i,j1,k1) = ...
do i = 1, m
  do j2 = 1, n
    do k2 = 1, p
      ... = a(i,j2,k2)
```

(b)

Figure 2.19: Loop Fission - (a) Before & (b) After Fission

available for locality enhancement. A large number of alternative approaches have been developed by researchers over the last decade to remove the limitations suffered by tiling — the goal has been to reason about transformations of non-perfectly nested loops and extending the notion of tiling to non-perfectly nested loops.

In the context of non-perfectly nested loops, five transformations have been studied extensively in helping improve locality:

```
do i = 1, m
  do j1 = 1, n
    a(i,j1) = ...
    b(i,j1) = ...
  do j2 = 1, n
    ... = b(i,j2)
```

(a)

```
do i = 1, m
  do j1 = 1, n
    a(i,j1) = ...
do i = 1, m
  do j = 1, n
    b(i,j1) = ...
do i = 1, m
  do j2 = 1, n
    ... = b(i,j2)
```

(b)

```
do i = 1, m
  do j1 = 1, n
    a(i,j1) = ...
do i = 1, m
  do j = 1, n
    b(i,j) = ...
    ... = b(i,j)
```

(c)

Figure 2.20: (a) Initial Program, (b) After Fission, (c) After Fusion

```
do i = 1, 100
  a(i) = ...
  do j = i, 50
    b(i,j) += a(i)
```

(a)

```
do i1 = 1, 50
  a(i1) = ...
  do j = i, 50
    b(i1,j) += a(i1)
do i2 = 51, 100
  a(i2) = ...
  do j = i, 50
    b(i2,j) += a(i2)
```

(b)

Figure 2.21: Index Set Split — (a) Before, (b) After

1. *Loop Fusion*: Loop fusion proceeds by interleaving the execution of iterations from loops in adjacent loop nests [49]. It is a basic transformations that works *across* loop nests. If the same array is used in two adjacent loops, loop fusion improves program performance by bringing iterations using the same array element close in time. Figure 2.18 shows the loop fusion transformation where the doubly nested `j1, k1` loop is fused with a doubly nested `j2, k2`

```
do i = 1, n
  do j = 1, m
    S1(i,j)
  do k = 1, p
    S2(i,k)
```

(a)

```
do i = 1, n
  do j = 1, m
    S1(i,j)
  if (j == m)
    do k = 1, p
      S2(i,k)
```

(b)

```
do i = 1, n
  do j = 1, n
    S1(i,j)
    do k = 1, p
      if (j == m)
        S2(i,k)
```

(c)

```
do i = 1, n
  do j = 1, n
    do k = 1, n
      if (k == 1)
        S1(i,j)
      if (j == m)
        S2(i,k)
```

(d)

Figure 2.22: Code sinking for non-SNL loop nests (a) – (d)

loop. Loop fusion typically requires that the loops being fused have identical loop bounds, otherwise guards need to be introduced that each statement in the initial program is executed the appropriate number of times.

In addition to directly enhancing locality, loop fusion can also be useful by converting a non-perfectly nested loop to a perfectly nested loop, after which the linear loop transformations as well as loop tiling can be applied to further improve program performance.

Loop fusion is not always legal. It is quite straightforward to determine whether a given loop fusion transformation is legal. The choice of loops chosen to be fused has an impact on which linear loop transformations can be applied next, and this relationship cannot always be predicted in advance.

```
do i = 1, n              do l = 1, n              do j = 1, n
  do j1 = 1, n             do j1 = 1, n             do i1 = 1, n
    S1: A(i,j1) = 0          S1: A(l,j1) = 0          A(i1,j) = 0
  do j2 = 1, n             do i = 1, n              do i2 = 1, n
    S2: B(i,j2) = 0          S2: B(i,j2) = 0          A(i2,j) = 0
```

(a) (b) (c)

Figure 2.23: (a) Initial code, (b) & (c) Two possible choices to interchange `i` & `j2`

2. *Loop Fission*: Loop fission, also known as *loop distribution*, is the converse of loop fusion. Loop fission by itself does not improve locality. However, as Figure 2.19 demonstrates, loop fission can be used to transform a non-perfectly nested loop into a perfectly nested loop, after which the full array of linear loop transformations can be applied. In addition, as Figure 2.20 demonstrates, loop fission can be followed by loop fusion to improve locality in the initial program by reorganizing the structure of loops and statements in the program. This sequence of fission and fusion is better than simply applying fusion to the `j1` and `j2` loops in initial program — accesses to arrays `A` and `B` are separated using the two-step process, resulting in lower interference between accesses to `A` and `B`.

   Loop fission is not always legal, but testing for legality of a fission operation is quite straightforward. As in the case of fusion, the choice of the loop fissioned has an impact on which subsequent transformations can be applied subsequently; this relationship cannot be predicted in advance.

3. *Index Set Splitting*: Index set splitting is a transformation on a single loop,

```
do k = 1, NMAX
  do i = k+1, NMAX
    do j = k+1, i
      if (i == k+1) && (j == k+1)
        a(k,k) = dsqrt (a(k,k))
      if (j == k+1)
        a(i,k) = a(i,k) / a(k,k)
      a(i,j) -= a(i,k) * a(j,k)
```

(a)

```
do k = 1, NMAX
  do i1 = k+1, NMAX
    do i2 = k+1, NMAX
      do j = k+1, i2
        if ((i1 == k+1) &&
            (i2 == k+1) &&
            (j == k+1)
          a(k,k) = dsqrt (a(k,k))
        if ((i2 == k+1) &&
            (j == k+1))
          a(i1,k) = a(i1,k) / a(k,k)
        if (i1 == NMAX)
          a(i2,j) -= a(i2,k) * a(j,k)
```

(b)

Figure 2.24: Code sinking of Cholesky, `kij` version – (a) fused, (b) distributed

where its range of iterations is split into two ranges. The net effect is to create two copies of the loop where the first copy enumerates over the first range and the second copy enumerates over the second range, as shown in Figure 2.21. Index set splitting by itself does not improve locality; however, as will be discussed shortly, researchers have realized that index set splitting can be used as an enabling step for loop fission, which is in turn an enabling transformation for other locality enhancing transformations. In addition, index set splitting can under certain conditions give rise to perfectly nested loops, after which linear loop transformations and tiling can be applied. For example, in Figure 2.21, it can be seen that the `j` loop contained within the `i2` loop is never executed(`i2` is always greater than 50) and can be eliminated. This transforms the `i2` loop into a perfectly nested loop.

Index set splitting is always legal. However, while performing an index set split, the split point in the initial range must be chosen carefully, as it determines what transformations can be applied subsequently.

4. *Code Sinking*: Code sinking, as has already been mentioned in Section 2.2.1, is the process of moving all statements into the innermost position, while introducing guards to ensure that all the statements are executed the correct number of times. Code sinking is performed for each statement not nested inside the innermost loop, one statement at a time.

   Code sinking for Singly Nested Loops (SNLs) is quite straightforward and has been dealt with in detail in the literature. However, code sinking for even simple non-SNL program leads to nonintuitive behavior. For example, when code sinking is applied to simple program in Figure 2.22(a), the resulting loop nest is nested three deep, even though both statements in the initial program are only nested within two loops. For more complex non-perfectly nested loops, the resulting perfectly nested loop is correspondingly more complex and the behavior of code sinking more unpredictable. For example, applying sinking to the fused variant of the `kij` version of Cholesky Factorization yields the code in Figure 2.24(a), whereas applying sinking the distributed variant of the same `kij` version of Cholesky Factorization yields Figure 2.24(b). This illustrates how the output of code sinking can change significantly with even a small change in the input code.

5. *Non-Perfect Strip-mine and Interchange*: One possible implementation of tiling of a loop in a perfectly nested loop is as a sequence of two transfor-

mations - *strip-mine* followed by *interchange*. Attempts have been made to extend this notion of tiling directly to non-perfectly nested loops. Strip-mining a loop does not depend on whether the loop is part of a perfectly nested loop. However, the notion of loop interchange for non-perfectly nested loops is not well defined. For example, in Figure 2.23(a), attempting to permute the `i` and `j2` loops has a simple meaning for statement S2 contained inside them. However, the situation for statement S1 nested inside the `i` and `j1` loops is different. Since `S1` is not nested within both `i` and `j1` loops, there are two possible choices on how the interchange of `i` and `j2` loops affects S1. One solution is to have all statement instances of `S1` executed in the same order as before. This corresponds to the code in Figure 2.23(b). An alternative solution is to reorder the statement instances of S1 as if loops `i` and `j1` had been permuted as well. This code, shown in Figure 2.23(c), clearly has different semantics and legality criteria from the code in Figure 2.23(b). It should be noted that this situation does not arise in the case of singly nested loops, since `j1` and `j2` loops could not both be nested inside the `i` loop if this program fragment were singly nested.

Even such a simple program as in Figure 2.23, non-perfectly nested loop interchange is non-trivial because of the set of choices presented. For more complex non-perfectly nested programs, where the desired final transformation often requires the cumulative effect of several non-perfectly nested loop interchange operations, the situation quickly becomes unmanageable.

```
do k = 1, N-1
  Scale step
  do i = k+1, N
    A(i,k) = A(i,k) / A(k,k)
  Update Step
  do j = k+1, N
    do i = k+1, N
      A(i,j) -= A(i,k)*A(k,j)
```

(a)

```
do k = 1, N-1, ks
  do kk = k, min(k+ks-1,N-1)
    Scale Step
    do i = kk+1, N
      A(i,k) = A(i,k)/A(k,k)
    Update Step
    do j = kk+1, N
      do i = kk+1, N
        A(i,j)-= A(i,kk)*A(kk,j)
```

(b)

```
do k = 1, N-1, ks
  do kk = k, min(k+ks-1,N-1)
    Scale Step
    do i = kk+1, N
      A(i,k) = A(i,k)/A(k,k)
    Update Step
    do j = kk+1, min(k+ks-1,N)
      do i = kk+1, N
        A(i,j)-= A(i,kk)*A(kk,j)
    Update Step
    do j = k+ks, N
      do i = kk+1, N
        A(i,j)-= A(i,kk)*A(kk,j)
```

(c)

```
do k = 1, N-1, ks
  do kk = k, min(k+ks-1,N-1)
    Scale Step
    do i = kk+1, N
      A(i,k) = A(i,k)/A(k,k)
    Update Step
    do j = kk+1, min(k+ks-1,N)
      do i = kk+1, N
        A(i,j)-= A(i,kk)*A(kk,j)
  do kk = k, min(k+ks-1,N-1)
    Update Step
    do j = k+ks, N
      do i = kk+1, N
        A(i,j)-= A(i,kk)*A(kk,j)
```

(d)

Figure 2.25: Blocking of LU Factorization (no pivoting) using Carr's Approach

```
do j = 1, NMAX
  do k = 1, j-1
    do i = j, NMAX
      a(i,j) -= a(i,k)*a(j,k)
  a(j,j) = dsqrt (a(j,j))
  do i = j+1, NMAX
    a(i,j) = a(i,j)/a(j,j)
```

(a)

```
do j = 1, NMAX
  do k = 1, j-1
    do i = j, NMAX
      a(i,j) -= a(i,k)*a(j,k)
  do i = j, NMAX
    if (i == j)
      a(j,j) = dsqrt(a(j,j))
    if (i > j)
      a(i,j) = a(i,j)/a(j,j)
```

(b)

```
do j = 1, NMAX
  do k = 1, j
    if (k < j)
      do i = j, NMAX
        a(i,j) -= a(i,k)*a(j,k)
    if (k == j)
      do i = j, NMAX
        if (i == j)
          a(j,j) = dsqrt(a(j,j))
        if (i > j)
          a(i,j) = a(i,j)/a(j,j)
```

(c)

```
do j = 1, NMAX
  do k = 1, j
    do i = j, NMAX
      if (k < j)
        a(i,j) -= a(i,k)*a(j,k)
    do i = j, NMAX
      if (k == j)
        if (i == j)
          a(j,j) = dsqrt(a(j,j))
        if (i > j)
          a(i,j) = a(i,j)/a(j,j)
```

(d)

Figure 2.26: Sinking & Fusion for Cholesky Factorization, `jki` version: (a) – (d)

```
do j = 1, NMAX
  do i = j, NMAX
    do k = 1, j-1
      a(i,j) -= a(i,k)*a(j,k)
  a(j,j) = dsqrt (a(j,j))
  do i = j+1, NMAX
    a(i,j) = a(i,j)/a(j,j)
```

(a)

```
do j = 1, NMAX
  do i = j, NMAX
    do k = 1, j-1
      a(i,j) -= a(i,k)*a(j,k)
  do i = j, NMAX
    if (i == j)
      a(j,j) = dsqrt(a(j,j))
    if (i > j)
      a(i,j) = a(i,j)/a(j,j)
```

(b)

```
do j = 1, NMAX
  do i1 = j, NMAX
    do k = 1, j-1
      a(i1,j) -= a(i1,k)*a(j,k)
    if (i1 == NMAX)
      do i2 = j, NMAX
        if (i2 == j)
          a(j,j) = dsqrt(a(j,j))
        if (i2 > j)
          a(i2,j) = a(i2,j)/a(j,j)
```

(c)

```
do j = 1, NMAX
  do i1 = j, NMAX
    do k = 1, j-1
      a(i1,j) -= a(i1,k)*a(j,k)
    do i2 = j, NMAX
      if (i1 == NMAX)
        if (i2 == j)
          a(j,j) = dsqrt(a(j,j))
        if (i2 > j)
          a(i2,j) = a(i2,j)/a(j,j)
```

(d)

Figure 2.27: Sinking & Fusion for Cholesky Factorization, `jik` version: (a) – (d)

2.3.1 Summary of Research Approaches

The program transformations described in Section 2.3 have been combined in various ways by researchers to address the problem of exploiting locality in non-perfectly nested loops. This section describes how these approaches perform on factorization routines from LAPACK. Three main approaches are of interest.

McKinley, Carr and Tseng [32] apply a series of loop transformations including loop fusion, loop distribution and the linear loop transformations (interchange, skewing and reversal) to improve memory access behavior of a program and thereby improve performance. Cholesky Factorization is discussed, but LU Factorization with Pivoting and QR Factorization are not. In the case of Cholesky Factorization, the authors' strategy for choosing transformations and the cost model suggest that the `kji` version of Cholesky Factorization has the best memory access, followed by `jki`, `kij`, `ikj`, `jik` and `ijk` versions respectively. As a result, the authors are able to derive the `kji` version starting with the fused variant of the `kij` version by applying a combination of loop distribution and loop permutation.

The experiments conducted on the SGI Octane as part of this dissertation do not bear out the conclusion reached by McKinley et al. Indeed, as Figure 2.14 and Figures 2.8 through 2.13 demonstrate, for small to medium problem sizes, the `jik` and `jki` versions significantly outperform all the other versions. Even for larger problems, these versions outperform all the other versions with the single exception of the fused variant of the `kij` version. Further analysis is complicated by the fact that the problem size is unknown in their results, and it is not clear if the performance results presented contain the effect of any other transformations performed by the native compiler (the compilation flags are not presented).

A very important reason for the *jik* and *jki* versions to outperform all the other versions is that in addition to exploiting locality in the Level 2 cache, locality in registers must be exploited as well. Indeed, for small or medium sized problems, registers are much more important, since the problem fits in the cache under these conditions. This suggests that dealing with locality for caches in isolation is likely to be counterproductive in general.

A more elaborate and sophisticated approach has been developed by Carr and Lehoucq [11, 9]. Their approach has been to attempt to perform a combination of imperfectly nested strip-mine interchange in addition to loop distribution and index set splitting to convert a non-perfectly nested loop into a program which is amenable to subsequent transformations. Figure 2.25 demonstrates their approach for LU Factorization with no pivoting. Figure 2.25(a) contains the input code for LU Factorization with no pivoting. The first step is to strip-mine the outer `k` loop. This produces the code in Figure 2.25(b). At this stage, an attempt is made to interchange the `kk` loop into the innermost position. However, since this would require imperfectly-nested loop interchange, an attempt is first made to distribute the `kk` loop around its constituent statements. However, this distribution cannot be performed as it violates dependences. To enable any distribution, it is then determined using array section analysis that the `j` loop of the *update step* must be index-set-split at the point `k+ks`, so that distribution can in fact be performed. This produces the code in Figure 2.25(c). At this stage the `kk` loop can in fact be distributed around the second copy of the `j` loop of the update step. Doing this produces the code in

Figure 2.25(d). At this stage, the `kk` loop surrounding the second update step can in fact be permuted into the innermost position, which achieves the overall tiling effect desired. After this, additional transformations such as unroll-and-jam are applied to optimize this code for registers and achieve computation and memory balance on superscalar hardware [10, 25].

The authors suggest that this approach extends naturally to Cholesky Factorization, and demonstrate the approach for the distributed variant of the `kij` version. However, it is not clear from the discussion if this approach works for all the seven versions of Cholesky Factorization. In the case of LU Factorization with Pivoting, the presence of pivoting introduces dependences preventing the distribution step analogous to that carried out to transform the code in Figure 2.25(c) to that in Figure 2.25(d). The authors suggest that this limitation be removed by introducing pattern matching into a compiler to recognize that the row-permutation step in LU Factorization with Pivoting commutes with the update step. In each of these three cases, by applying the transformations the authors suggest, it is possible to derive the handwritten library routines for these codes implemented in LAPACK. Finally, in the case of QR Factorization with Householder Reflections, the authors apply a combination of index set splitting, loop distribution and loop permutation to determine a blocked version which is different from the handwritten code in LAPACK. The LAPACK implementation uses information about matrix product associativity to derive a blocked algorithm which is materially different from the unblocked version of QR Factorization using Householder Reflections. The authors do not suggest a mechanism for providing information about matrix product associativity to the compiler.

Carr and Lehoucq's approach certainly addresses the question of deriving automatically blocked versions of matrix factorization codes found in LAPACK, and in fact produces code competitive with LAPACK code. However, as the authors recognize, a sequence of *nine* compiler transformations including index set splitting, loop distribution, unroll-and-jam, pattern matching for pivoting and strip-mine-and-interchange needs to be applied to the input code to achieve this. This leads to a significant phase-ordering problem and makes this approach unsuitable in a production setting where it is critical to reason about the cost and benefit of program transformations for a very large number of programs. The necessity of applying many complex transformations in conjunction makes it very hard to determine a simple cost metric regarding the net benefit at the end of the day.

Finally, a third approach to developing handwritten algorithms in LAPACK has been developed by Ramanujam and Schreiber. Their approach is to use a combination of code sinking and loop fusion to convert a non-perfectly nested loop into a perfectly nested loop, which can then be tiled. As Figure 2.24(a) demonstrates, the fused variant of the `kij` version of Cholesky Factorization can be converted into a perfectly nested loop using code sinking alone, and all the three loops in the resulting loop nest are tileable. However, the distributed variant of the same `kij` version is transformed into a four-deep loop nest, in which only the two innermost loops are tileable, on the application of code sinking alone. Recognizing this, Ramanujam and Schreiber have suggested that loop fusion be used in conjunction with the code sinking process to eliminate this scenario. For example, in the distributed variant of the `kij` version of Cholesky Factorization, the scale and update loops can be fused to yield the fused variant of the `kij` version, after which code sinking can be applied. The author of this dissertation has explored this approach with Schreiber [44, 43] for the other versions of Cholesky Factorization. Figure 2.26 demonstrates how code sinking and fusion can be used in conjunction to convert the `jki` version of Cholesky Factorization to a perfectly nested loop. Figure 2.26(a) shows the input code. Figure 2.26(b) is obtained from this code by sinking the square root step downwards into the scale loop. The resulting `i` loop can now be sunk upwards into the `k` loop to yield Figure 2.26(c). In this code, the if statement guards surrounding both the `i` loops can be permuted with the corresponding `i` loops into the innermost position to yield the code in Figure 2.26(d). Both the `i` loops can be fused in this code to yield a three-deep loop nest which is fully tileable.

Complicated as this sequence of steps is, it does not always work. For example, trying the same sequence of steps on the `jik` version of Cholesky Factorization yields the code in Figure 2.27(d). However, in this code, fusion of the `k` and `i2` loops is illegal and nothing further can be done to convert this code into a perfectly nested loop three deep. To avoid this scenario, either the `i` and `k` loops surrounding the update step in Figure 2.27(a) must be permuted to revert to the `jki` version, or the two `i` loops in Figure 2.27(b) must be fused together. Thus, the sequence of steps of code sinking and fusion is very different even between two very similar versions of Cholesky Factorization, obtained from each other simply by loop interchange. It remains an open research problem today to discover a systematic approach to determining the “correct” sequence of code sinking and fusion to be able to obtain a three deep perfectly nested loop starting with each of the seven versions of Cholesky Factorization.

2.4 Summary of the Problem

In summary, it is clear that as memory hierarchies get deeper, exploiting locality is crucial to keeping the processor fully utilized and achieving high performance. Unfortunately, most algorithms in the dense linear algebra domain in do not exploit locality in their natural form and perform poorly. Hand-written libraries alleviate this problem somewhat by providing portable implementations of common computations. In the context of dense linear algebra, libraries have followed a two-stage layered approach - (i) the BLAS library provided by individual vendors encapsulating core computations such as matrix multiplication and matrix vector product, and (ii) the LAPACK library containing hand-restructured algorithms layered on top of the BLAS library. However, the libraries involve significant development and maintenance effort and are specific to the dense linear algebra domain. This raises the question of developing compiler techniques using which it would be possible to achieve the performance of handwritten libraries using program automatic program restructuring techniques.

The state-of-the-art in compiler technology for locality enhancement in this domain is based on the linear loop transformation framework for perfectly nested loops. Perfectly nested loops are simple programs in which all the assignment statements are completely enclosed within all the loops. Two key loop transformations, *inter-*

change and *tiling*, allow a perfectly nested loop with poor locality to be transformed into a program with improved locality. The core computations in the BLAS library are perfectly nested, and for these programs, compiler technology is very effective - for example in the case of matrix multiplication, a combination of loop interchange and tiling can achieve 80% of the performance of the handwritten code. However, the more complex algorithms in LAPACK are all non-perfectly nested, and there is no systematic framework to convert a non-perfectly nested loop to a perfectly nested loop. Various ad-hoc strategies have been proposed, but their ad-hoc nature makes them unsuitable for integration in a production setting. As a result, existing production compiler technology is not very effective for computations in LAPACK, and handwritten library code is *3-5 times* faster than what the compiler can achieve. This is an undesirable situation.

Chapter 3

Data-centric Shackling

From the performance results in Section 2.2, it should be clear that there is a considerable gap between what state-of-the-art compiler technology can achieve on codes from dense linear algebra and the corresponding hand tuned library codes. In particular, while the singly nested loop-based tiling framework is very effective for codes implemented in the BLAS library, it is completely inadequate for computations that are found in an application level library such as LAPACK. In this chapter, I present novel compiler technology developed as part of this dissertation which allows one to overcome the limitations of tiling.

3.1 Data-centric Transformations

Since the goal of program transformation is to enhance data reuse and reduce data movement through the memory hierarchy, it would seem advantageous to have a tool that orchestrates data movement *directly*, rather than as a side-effect of control flow

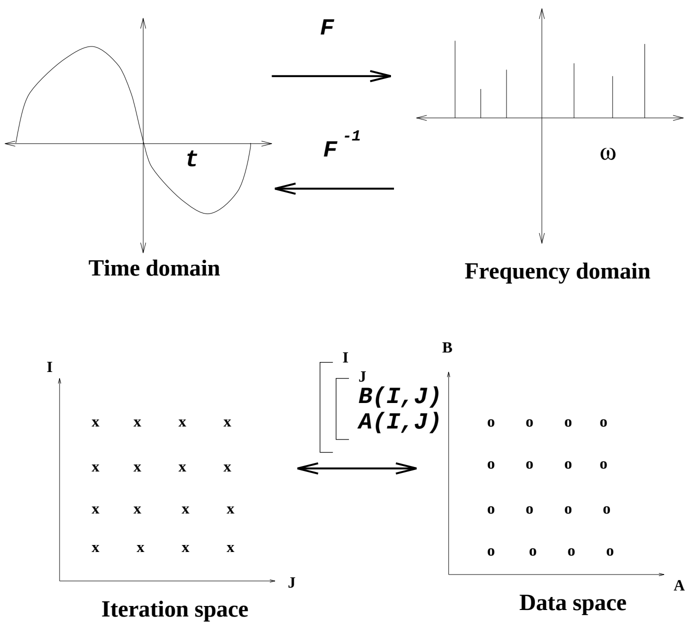


Figure 3.1: Analogy between Locality Enhancement & Signal Processing

manipulations. The ultimate result of the orchestration is, of course, a transformed program with the desired data reuse, but to get that program, the tool would reason directly about the desired data flow rather than about the control flow of the program. An analogy can be drawn with signal processing(Figure 3.1) - the input and the output of signal processing is a signal that varies with time, and in principle, all processing can be done in the time domain. For example, if one wanted to remove all the high frequency components of a signal, it is certainly a problem one could solve in the time domain. However, it is often more convenient to take a Fourier transform of the signal, formulate a solution in the frequency domain, and then apply an inverse Fourier transform to map the solution back into the time domain. The rest of this chapter argues that for locality-enhancing transformations, one should focus on the data domain to obtain desirable transformations. While equivalent transformations can also be obtained by concentrating on the control flow, focusing on the data provides a more natural solution to the problem. In other words, this chapter argues that a *data-centric* approach is more natural for locality enhancement than a *control-centric* one.

The key mechanism that allows a data-centric approach to program transformations is a *data shackle*. The key components of the data shackle and how it can be used to enhance locality are described in Section 3.2. The ease with which the data-centric approach tackles the problem of locality has enabled it to be integrated into the production compiler at Silicon Graphics. The particulars of this integration are dealt with in Section 3.3. Section 3.4 presents a detailed evaluation of this new technology on core computations from LAPACK.

3.2 Data Shackle

Any program transformation framework which reorders the execution of statements in the initial program needs a mechanism to specify the desired reordering. In the data-centric world, this mechanism will be called the *Data Shackle*.

Data shackles will be defined for the following class of programs: any sequence of nested loops (perfectly or non-perfectly) are considered. The only constraint imposed is that loop bounds be affine functions of surrounding loop indices and constant parameters. In addition, conditionals can be present in the loop body. In other words, data shackles can be defined for a very large class of programs, easily including the algorithms implemented in LAPACK.

Definition 1 *A* shackleable program *is a series of assignment statements surrounded by loops and conditionals. All loop bounds must be affine functions of surrounding loop indices and constant parameters.*

Definition 2 *For statement* `s`, l(s) *represents the loops surrounding* `s` *in the program. (*`S`*,$\underline{i}$) represents a single execution of* `s`*, where $\underline{i}$ is a vector of integers corresponding the values of the loop index variables in* l(s) - *the left to right order of the entries in $\underline{i}$ corresponds to outer-to-inner ordering of loops in* l(s).

Definition 3 *A single execution of* `s` *is also referred to as a* dynamic statement instance *of s, represented by (*`s`*, $\underline{i}$) as described in definition 2.*

3.2.1 The Data Shackle

Given the definitions above, the notion of a data shackle can now be defined.

Definition 4 *A* data shackle *is a specification in three parts.*

- *A* partitioning *of a data structure into blocks.*
- *An order of enumeration of the resulting blocks.*
- *An assignment of dynamic statement instances to blocks - note that this step simply assigns statement instances to blocks, without specifying an order of execution for these statement instances.*

Each of the steps is now examined now in more detail:

- The data objects of interest are multidimensional arrays. An array is partitioned into blocks using multiple sets of parallel equidistant cutting planes. The *cutting planes matrix* is defined as the matrix whose columns are the normals to the different sets of cutting planes; the order of these columns is determined by the order in which the sets of cutting planes are applied to partition the array. Figure 3.2 shows the blocking of a two-dimensional array with two sets of cutting planes; the cutting planes matrix is $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. It is worth noting that this partitioning is only a logical partitioning and does not imply a change in the storage of the array in memory.
- A block is assigned the *block co-ordinate* $(x_1, \ldots, x_m)$ if it is bounded by cutting planes numbered $x_i - 1$ and x_i from the i^{th} set of cutting planes. The order of enumeration of blocks is defined to be the lexicographic order of the block coordinates.
- The last step is the assignment of dynamic statement instances to blocks. From each statement `s`, a *single* reference `r` to the partitioned array is chosen. Whenever a block of data is visited in the enumeration, all dynamic instances of `s` are performed for which the data touched by reference `r` is contained in the current block[1]. As mentioned before, the order in which the statement instances assigned to a given block are performed is left unspecified at this stage. `R` is referred to as the *data-centric reference* for `s`.

[1]because of other data references in statement `s`, these statement instances may touch data outside that block.

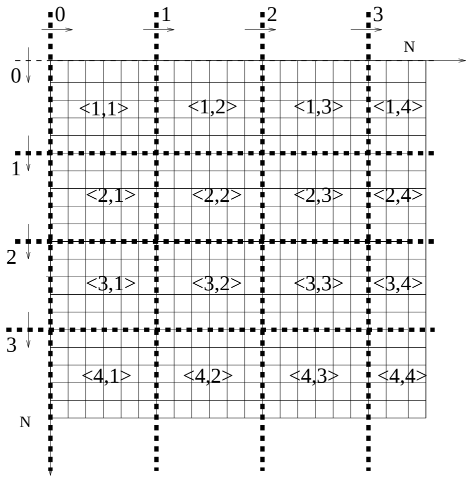


Figure 3.2: Cutting Planes on a two-dimensional array

The rationale for the term "data shackle" should now be clear. One thinks of an instrument like a pantograph in which a master device visits the blocks of data in lexicographic order, while a slave device shackled to it is dragged along some trajectory in the iteration space of the program. It is also convenient to be able to refer to statement instances "shackled" to a particular block of data.

The matrix multiplication code of Figure 3.3(i) is an example of a data shackle. Matrix `c` is partitioned as shown in Figure 3.2, using two sets of cutting planes, one set parallel to each of the axes. In each set, the cutting plane separation is 25, yielding a partitioning of `c` into blocks of 25× 25. The cutting plane matrix for `c` is given by $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. There is a single statement `s1`, for which `c(i,j)` is picked as the data-centric reference. Blocks of `c` are enumerated one block row at a time from top to bottom, and within a block row, from left to right. When a given block of `c`

```
do i = 1, n
  do j = 1, n
    do k = 1, n
      S1: C(i,j) += A(i,k)Bb(k,j)
```

(i) Matrix Multiplication

```
do k = 1, n
  S1: A(k,k) = sqrt (A(k,k))
  do i = k+1, n
    S2: A(i,k) = A(i,k) / A(k,k)
  do i = k+1, n
    do j = k+1, i
      S3: A(i,j) -= A(i,k)*A(j,k)
```

(ii) Cholesky Factorization

Figure 3.3: Examples: (a) Matrix Multiplication and (b) Cholesky Factorization

is visited, only those instances of `s1` are performed for which the data touched by the `c(i,j)` reference lie within the current block. The order of execution of these statement instances is not mandated by the shackle, but a simple choice is program order in the source program.

A data shackle yields a specification of the desired execution of statement instances. Naive code for executing this specification is shown in Figure 3.4. The two outer loops iterate over the blocks of `c`. For every block of `c`, the entire original iteration space is visited, and every iteration is tested to see if it should be executed - if the data touched by `c(i,j)` lies within the current block, the iteration is executed, otherwise it is not. One can think of this test as a pair of conditionals introduced to be nested tightly around `s1`. The conditionals *localize* the data touched by the data-centric reference to be within the current block, and will be referred to as *localization constraints*.

The code in Figure 3.4 is not very efficient, and is similar to *runtime resolution* code generated when shared-memory programs are compiled for distributed-memory machines [40]. Fortunately, since the conditionals are affine conditions on

```
do b1 = 1 ,   ⌈(n/25)⌉
  do b2 = 1 ,   ⌈(n/25)⌉
    do i = 1 ,  n
      do j = 1 ,  n
        do k = 1 ,  n
          if ((b1-1)*25 < i <= b1*25) &&
              ((b2-1)*25 < j <= b2*25)
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

Figure 3.4: Naive code to execute the specification of the data shackle for `c`

surrounding loop bounds, they can be simplified using standard polyhedral algebra. Section 3.3 presents a simple algorithm based on a solver for linear integer systems which allows one to do this simplification automatically to produce the code in Figure 3.5. Tools with similar capability, such as the Omega calculator [38] are also available in the public domain.

The net effect of the data shackle for `c` is the following: blocks of `c` are computed at a time by taking the product of a block row of `a` and a block column of `b`. This code is not the same as the blocked matrix matrix multiplication in [24] described in chapter 2. The shackle for the `c` array achieves good locality for `c` by constraining the `i` and `j` loop indices, but leaves the `k` loop unconstrained. This leads to good locality for `c`, but poor locality for the `a` and the `b` arrays. However, this issue can be addressed using the notion of *composition of shackles*

```
do t1 = 1 ,   ⌈(n/25)⌉
  do t2 = 1 ,   ⌈(n/25)⌉
    do it = (t1-1)*25 +1 ,  min(t1*25,n)
      do jt = (t2-1)*25 +1 ,  min(t2*25,n)
        do k = 1 ,  n
          C(it,jt) = C(it,jt) + A(it,k) * B(k,jt)
```

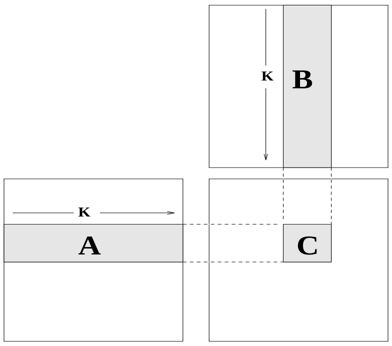


Figure 3.5: Simplified code produced by blocking C for matrix matrix multiply

3.2.2 Composition of Shackles

Figure 3.6 provides the intuition behind composition of shackles. The key is to visualize a single shackle as a reordering transformation. In Figure 3.6, ovals are used to group statement instances, which are represented by solid filled circles of different shades corresponding to different statements. A single shackle divides the initial set of statement instances into several ordered partitions. All statement instances in one partition are executed before the next partition is visited, however no policy is mandated on ordering of executions in the same partition. However, as the shackle for `C` in the matrix multiplication example of Figure 3.5 demonstrates, this is not sufficient to achieve complete locality for all the arrays in the program.

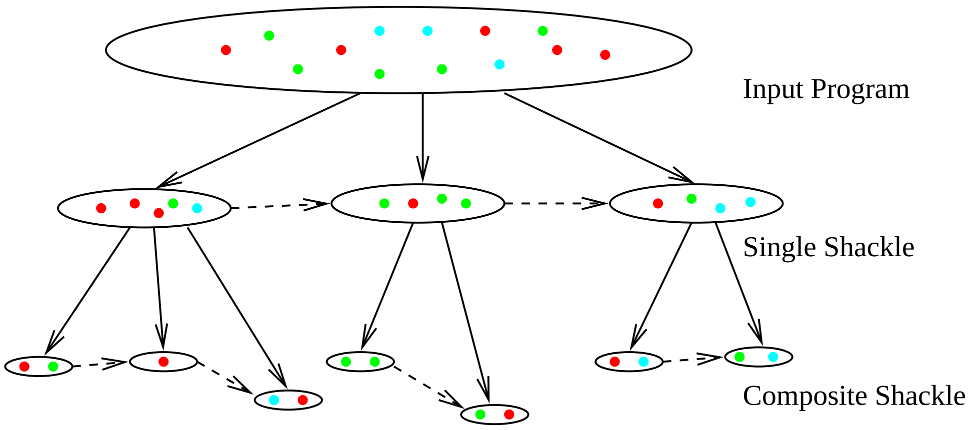


Figure 3.6: Composition as recursive partitioning of Statement Instances

The key insight is that just as the initial set of statement instances can be refined by a single shackle into groups of statement instances, each of these groups can be further refined into even smaller partitions using an additional shackle. The net effect is to provide a finer grain of control over the partitioning of statement instances, and hence improved locality.

An example makes the point: a two-level shackle for matrix multiplication can be formed as follows: As before, `C` can be partitioned into two-dimensional blocks, and `C(i,j)` can be chosen as the data centric reference for `S1`. Now, a second shackle can be chosen for (say) `A` by partitioning `A` into two-dimensional blocks, and choosing `A(i,k)` as the data centric reference for `S1`. The order of enumeration for the composite shackle is defined as follows: the blocks of `C` are visited in the order specified by the cutting planes for `C`, and for every visited block of `C`, all the blocks of `A` are visited in the order specified by the cutting planes on `A`. Finally, an instance of statement `S1` is executed only when the localization constraints for *both* `C` and `A` are satisfied - i.e `C(i,j)` lies in the block of `C` being visited and `A(i,k)` lies in the block of `A` being visited. Figure 3.7 shows naive code that could be used to implement this composite shackle for matrix multiplication. As in the case of a single shackle, this code is inefficient because of the conditionals nested inside loops. As before, these conditionals can be simplified automatically to yield the code in Figure 3.8. It should be noted that this simplified code is precisely the blocked version of matrix multiplication.

```
do c1 = 1 ,   ⌈(n/25)⌉
  do c2 = 1 ,   ⌈(n/25)⌉
    do a1 = 1 ,   ⌈(n/25)⌉
      do a2 = 1 ,   ⌈(n/25)⌉
        do i = 1 ,  n
          do j = 1 ,  n
            do k = 1 ,  n
              if ((c1-1)*25 < i <=c1*25 && (b2-1)*25 < j <=b2*25) &&
              if ((a1-1)*25 < i <=a1*25 && (a2-1)*25 < k <=a2*25)
                C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

Figure 3.7: Naive code to execute the a composite shackle for matrix multiplication

```
do c1 = 1 ,   ⌈(n/25)⌉
  do c2 = 1 ,   ⌈(n/25)⌉
    do a2 = 1 ,   ⌈(n/25)⌉
      do i = c1*25-24, min(c1*25,n)
        do j = c2*25-24, min(c2*25,n)
          do k = a2*25-24, min(a2*25,n)
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

Figure 3.8: Simplified code for the composite shackle in matrix multiplication

3.2.3 Shackling for Cholesky Factorization

The full power of data shackling can be demonstrated more adequately in the context of a non-perfectly nested loop program such as Cholesky factorization. In the Cholesky factorization code, array `A` can be shackled in a manner similar to Figure 3.2. When a block is scheduled, all statement instances that *write* to that block (in program order) can be performed. In other words, the reference chosen from each statement of the loop nest is the left hand side reference in that statement. Using polyhedral algebra methods, the code in Figure 3.9 can be automatically generated. In this code, data shackling regroups the iteration space into four sections as shown in Figure 3.10. Initially, all updates to the diagonal block from the left are performed (Figure 3.10(i)), followed by a *baby Cholesky factorization* [24] of the diagonal block (Figure 3.10(ii)). For each off-diagonal block, updates from the left (Figure 3.10(iii)) are followed by interleaved scaling of the columns of the block by the diagonal block, and local updates(Figure 3.10(iv)).

Note that just as in the case of matrix matrix product, this code is only partially blocked (compared to LAPACK code) — although all the writes are performed into a block when we visit it, the reads are not localized to blocks. Instead, the reads are distributed over the entire left portion of the matrix. As before, this problem is solved using composition of shackles. Both the shackles are defined for array `A` in the program, and divide up `A` into two-dimensional blocks the same way. The first shackle chooses `A(k,k)` from `S1`, `A(i,k)` from `S2` and `A(j,l)` from `S3`. The second shackle chooses `A(k,k)` from `S1`, `A(i,k)` from `S2` and `A(j,k)` from `S3`. The first shackle chooses references that write to the block, while the second shackle

```
do t1 = 1, (n+63)/64
  Update diag. block from left
  do t3 = 1, 64*t1-64
    do t6 = 64*t1-63, min(n,64*t1)
      do t7 = t6, min(n,64*t1)
        A(t7,t6)-=A(t7,t3)*A(t6,t3)

  Cholesky factor diag. block
  do t3 = 64*t1-63, min(64*t1,n)
    A(t3,t3) = sqrt(A(t3, t3))
    do t5 = t3+1, min(64*t1,n)
      A(t5,t3) = A(t5,t3)/ A(t3,t3)
    do t6 = t3+1, min(n,64*t1)
      do t7 = t6, min(n,64*t1)
        A(t7,t6)-=A(t7,t3)*A(t6,t3)

do t2 = t1+1, (n+63)/64
Apply updates from left to
off-diagonal block
  do t3 = 1, 64*t1-64
    do t6 = 64*t1-63, 64*t1
      do t7 = 64*t2-63,min(n,64*t2)
        A(t7,t6)-=A(t7,t3)*A(t6,t3)

  Apply internal scale/updates
  to off-diagonal block
  do t3 = 64*t1-63, 64*t1
    do t5 = 64*t2-63, min(64*t2,n)
      A(t5,t3) = A(t5,t3)/ A(t3,t3)
    do t6 = t3+1, 64*t1
      do t7 = 64*t2-63,min(n,64*t2)
        A(t7,t6)-=A(t7,t3)*A(t6,t3)
```

Figure 3.9: Data shackle applied to right-looking Cholesky factorization

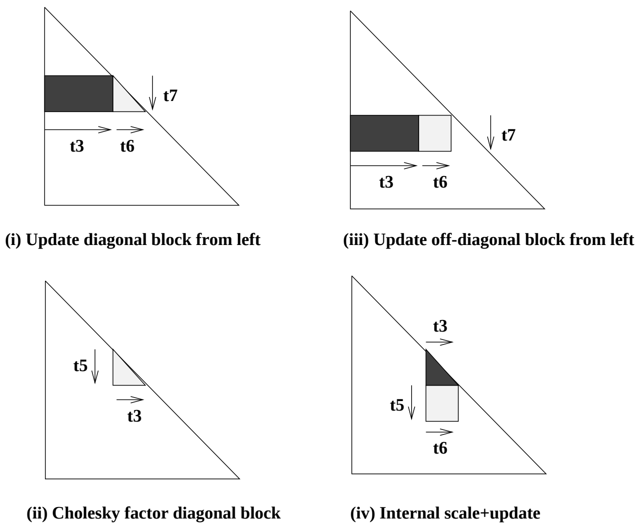


Figure 3.10: Pictorial View of Code in Figure 3.9

chooses references that read from the block. It can be shown quite easily that this composition in fact yields the fully blocked Cholesky factorization developed by hand in LAPACK.

3.2.4 Shackling and Locality

By shackling a data reference `R` in a source program statement `S`, it is ensured that the memory access made from that data reference at any point in program execution will be constrained to the "current" data block. Turning this around, one can see that when a block becomes current, all instances of statement `S` for which the reference `R` accesses data in that block are performed. Therefore, this reference enjoys perfect *self-temporal locality* [45]. Considering all shackled references together, it is easy to see that they enjoy perfect *group-temporal locality* for this set of references; of course, references outside this set may not necessarily enjoy group-temporal locality with respect to this set. As mentioned earlier, no order is mandated for statement instances assigned to a single block of data. However, if all dimensions of the array are blocked and the block fits in cache (or whatever level of the memory hierarchy is under consideration), *spatial locality* is exploited as well regardless of whether the array is stored in column-major or row-major order. An interesting observation is that if stride-1 accesses are mandated for a particular reference, a simple solution is to use cutting planes with unit separation enumerating the elements of the array in storage order. Enforcing stride-1 accesses *within* the blocks of a particular shackle can be accomplished by the composition of shackles.

3.2.5 Legality of a Shackle

Since data shackling reorders statement instances, it must be ensured that dependences are not violated. Let `(S1,i1)` and `(S2,i2)` be two dynamic statement instances such that there is a dependence from `(S1,i1)` to `(S2,i2)`. In addition, let these two instances be executed when blocks b_1 and b_2 are touched respectively. For the data shackle to be legal, either b_1 and b_2 must be identical, or b_1 must be touched before b_2. If so, the data shackle is said to *respect* that dependence. A data shackle is legal if it respects all dependences in the program. Since the application of a data shackle is not limited to perfectly nested loops like Cholesky factorization, it is not possible to use dependence abstractions like distance and direction to verify legality. Instead, the subsequent discussion formulates an integer linear programming problem for this purpose.

An Example

To understand the general algorithm, it is useful to consider first a simple example: in right-looking Cholesky factorization, the following discussion formulates the problem of ensuring that the flow dependence from the assignment of `A(k,k)` in `S1` to the use of `A(k,k)` in `S2` is respected by the data shackle from which the program of Figure 3.9 was generated[2]. The first step is to write down a set of integer inequalities that represent the existence of a flow dependence between an instance of `S1` and an instance of `S2`. Let `S1` write to an array location in iteration k_w of

[2]The shackle was produced by shackling the matrix `A` as shown in Figure 3.2, and choosing the left hand side references of all assignment statements in Figure 3.3(ii) for shackling.

$$\begin{cases} k_r = k_w, i_r = k_w & \text{(same location)} \\ n \geq k_w \geq 1 & \text{(loop bounds)} \\ n \geq k_r \geq 1 & \text{(loop bounds)} \\ n \geq i_r \geq k_r + 1 & \text{(loop bounds)} \\ k_r \geq k_w & \text{(read after write)} \end{cases} \tag{3.1}$$

Figure 3.11: Equations testing the existence of a Flow dependence

the k loop, and let `S2` read from that location in iteration (k_r, i_r) of the k and i loops. A flow dependence exists if the linear inequalities in Figure 3.11 have an integer solution [49]. Next, conditions are added expressing the situation that the instance of `S2` is performed when block (b_{11}, b_{12}) is scheduled, and the instance of `S1` is done when block (b_{21}, b_{22}) is scheduled. The final step is to add a condition that represents the situation where the dependence is violated in the transformed code. In other words, a condition is added stating that block (b_{11}, b_{12}) is "touched" strictly after (b_{21}, b_{22}). These conditions are represented in Figure 3.12.

If the conjunction of the two sets of conditions from Figure 3.11 and Figure 3.12 has an integer solution, it means that there is a dependence, and that dependent instances are performed in the wrong order. Therefore, if the conjunction has an integer solution, the data shackle violates the dependence and is not legal. This problem can be viewed geometrically as asking whether a union of certain polyhedra contains an integer point, and can be solved using standard polyhedral algebra.

This test can be performed for each dependence in the program. If no dependences are violated, the data shackle is legal.

$$
\begin{cases}
\text{Writing iteration done in } (b_{11}, b_{12}) \\
b_{11} * 25 - 24 \le k_w \le b_{11} * 25 \\
b_{12} * 25 - 24 \le k_w \le b_{12} * 25 \\
\text{Reading iteration done in } (b_{21}, b_{22}) \\
b_{21} * 25 - 24 \le k_r \le b_{21} * 25 \\
b_{22} * 25 - 24 \le i_r \le b_{22} * 25 \\
\text{Blocks visited in bad order} \\
(b_{11} < b_{21}) \vee ((b_{11} = b_{21}) \wedge (b_{12} < b_{22}))
\end{cases}
\tag{3.2}
$$

Figure 3.12: Equations representing violation of a flow dependence

3.2.6 Formal View of Data Shackles

The formulation of the general problem of testing for legality of a data shackle becomes simpler if the notion of shackling data is first generalized. A data shackling, such as the one shown in Figure 3.2, can be viewed simply as a map that assigns co-ordinates in some new space to every data element in the array. For example, if the block size in this figure is 25 x 25, array element (a_1, a_2) is mapped to the co-ordinate $((a_1\ div\ 25) + 1, (a_2\ div\ 25) + 1)$ in a new two-dimensional space. Note that this map is not one-to-one. The bottom part of Figure 3.13 shows such a map pictorially. The new space is totally ordered under lexicographic ordering. The data shackle can be viewed as traversing the remapped data in lexicographic order in the new co-ordinates; when it visits a point in the new space, all statement instances mapped to that point are performed.

Therefore, a data shackle can be viewed as a function **M** that maps statement

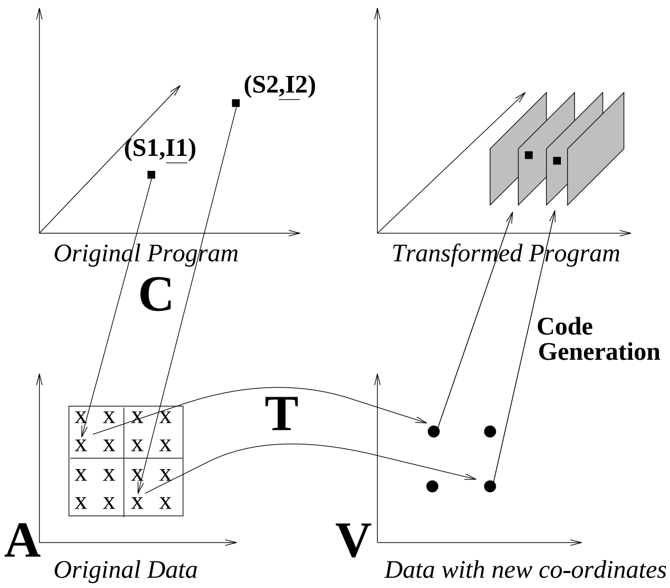


Figure 3.13: General view of data shackles

instances to a totally ordered set (`V`, $\prec$). For the shackling shown in Figure 3.13, `C:(S,I)` $\rightarrow A$ maps statement instances to elements of array `A` through data-centric references, and `T:A` $\rightarrow$ `V` maps array elements to block co-ordinates. Concisely, $\mathbf{M} = \mathtt{T}\circ\mathtt{C}$.

Given a function $\mathbf{M}$:`(S,I)`$\rightarrow$`(V,`$\prec$`)`, the transformed code is obtained by traversing `V` in increasing order, and for each element `v` $\in$ `V`, executing the statement instances $\mathbf{M}^{-1}$(`v`) in program order in the original program.

3.2.7 Legality for Shackles Revisited

Armed with the formal view of a data shackle, the conditions for the legality of a data shackle can now be stated concisely. Theorem 1 states necessary and sufficient conditions for a data shackle to preserve dependences in the initial program.

Theorem 1 *A map* $\mathbf{M}$*:*`(S,I)` $\rightarrow$`(V,`$\prec$`)` *generates legal code iff the following condition is satisfied for every pair of dependent statements* `S1` *and* `S2`*.*

- *Introduce vectors of unknowns* $\underline{\texttt{i1}}$ *and* $\underline{\texttt{i2}}$ *that represent instances of dependent statements* `S1` *and* `S2` *respectively.*
- *Formulate the inequalities that must be satisfied for a dependence to exist from instance* $\underline{\texttt{i1}}$ *of statement* `S1` *to instance* $\underline{\texttt{i2}}$ *of statement* `S2`*. This is standard [49].*
- *Formulate the predicate* $\mathbf{M}(\texttt{S2},\underline{\texttt{i2}}) \prec \mathbf{M}(\texttt{S1},\underline{\texttt{i1}})$*.*
- *The conjunction of these conditions does not have an integer solution.*

Proof: The proof is straightforward, hence omitted. □

Viewing shackling as a remapping of data co-ordinates simplifies the development of the legality test. This remapping is merely an abstract mathematical device to enforce a desired order of traversal through the array; *the physical array itself is not necessarily reshaped.* For example, in the blocked matrix multiplication code in Figure 3.5, array `C` need not be laid out in "block" order to obtain the benefits of shackling this array. Of course, nothing prevents shackling from reshaping the physical data array if the cost of converting back and forth from a standard representation is tolerable. Physical data reshaping has been explored by other researchers [15, 4].

It's worth noting at this stage that in the formal view of data shackles, there are no restrictions on how the mapping specified by the shackling is generated.

As long as such a mapping from old data-coordinates to new coordinates can be defined for each array being shackled, and a data-centric reference to that array defined for every statement, it is meaningful to think of the mapping induced by a shackle. In particular, the data-centric reference for a statement can in fact be a reference to the shackled array that occurs in a *different* statement. This removes the restriction that the shackled array must occur in every assignment statement, since any statement not containing a reference shackled array can choose a reference to the array from a different statement as its data-centric reference.

3.2.8 Composition of Shackles Revisited

The formal view of data shackles provides a very simple interpretation for the composition of shackles - it is simply a Cartesian product of the reordering imposed by individual shackles. A formal definition of the Cartesian product of data shackles is the following. From the discussion in Section 3.2.5, a data shackle for a program P can be viewed as a map $\mathbf{M}$:$(\mathtt{S},\mathtt{I}) \rightarrow \mathtt{V}$, whose domain is the set of statement instances and whose range is a totally ordered set.

Definition 5 *For any program P, let*

$$\begin{cases} \mathbf{M}_1 : (\mathtt{S},\mathtt{I}) \rightarrow \mathtt{V}_1 \\ \mathbf{M}_2 : (\mathtt{S},\mathtt{I}) \rightarrow \mathtt{V}_2 \end{cases}$$

be two data shackles. The Cartesian product $\mathbf{M}_1 \times \mathbf{M}_2$ *of these shackles is defined as the map whose domain is the set of statement instances, whose range is the Cartesian product* $\mathtt{V}_1 \times \mathtt{V}_2$ *and whose values are defined as follows: for any statement instance* $(\mathtt{S},\underline{\mathtt{i}})$,

$(\mathbf{M}_1 \times \mathbf{M}_2)(\mathtt{S}, \underline{\mathtt{i}}) \; = < \mathbf{M}_1(\mathtt{S}, \underline{\mathtt{i}}), \mathbf{M}_2(\mathtt{S}, \underline{\mathtt{i}}) >$

The product domain $\mathtt{V}_1 \times \mathtt{V}_2$ of two totally ordered sets is itself a totally ordered set under standard lexicographic order. Therefore, the code generation strategy and associated legality condition are identical to those in Section 3.2.5. It is easy to see that for each point $\underline{v_1} \times \underline{v_2}$ in the product domain $\mathtt{V}_1 \times \mathtt{V}_2$, we perform the statement instances in the set $(\mathbf{M}_1 \times \mathbf{M}_2)^{-1}(\underline{v_1}, \underline{v_2}) = \mathbf{M}_1{}^{-1}(\underline{v_1}) \cap \mathbf{M}_2{}^{-1}(\underline{v_2})$.

In the implementation, each term in an n-ary Cartesian product contributes a guard around each statement. The conjunction of these guards determines which statement instances are performed at each step of execution. Therefore, these guards still consist of conjuncts of affine constraints. As with single data shackles, the guards can be simplified using standard polyhedral algebra techniques discusses in Section 3.3.

The product of two shackles is always legal if the two shackles are legal by themselves. However, a product $\mathbf{M}_1 \times \mathbf{M}_2$ can be legal even if $\mathbf{M}_2$ by itself is illegal. This is analogous to the situation in loop nests where a loop nest may be legal even if there is an inner loop that cannot be moved to the outermost position; the outer loop in the loop nest "carries" the dependence that causes difficulty for the inner loop.

Examples

In matrix multiplication, it is easy to see that shackling any of the three references (`C(i,j)`,`A(i,k)`,`B(k,j)`) to the appropriate blocked array is legal. Therefore, all Cartesian products of these shackles are also legal. The Cartesian product $\mathbf{M}_C \times \mathbf{M}_A$

of the C and A shackles produces the code in Figure 3.8. It is interesting to note that further shackling with the B shackle (that is the product $\mathbf{M}_C \times \mathbf{M}_A \times \mathbf{M}_B$) does not change the code that is produced. This is because shackling `C(i,j)` to the blocks of C and shackling `A(i,k)` to blocks of A imposes constraints on the reference `B(k,j)` as well. A similar effect can be achieved by shackling the references `C(i,j)` and `B(k,j)`, or `A(i,k)` and `B(k,j)`.

A more interesting example is the Cholesky code. In Figure 3.3(ii), it is easy to verify that there are six ways to shackle references in the source program to blocks of the matrix (choosing `A(k,k)` from statement `S1`, either `A(i,k)` or `A(k,k)` from statement `S2` and either `A(j,l)`, `A(j,k)` or `A(l,k)` from statement `S3`). Of these, only two are legal: choosing `A(k,k)` from `S1`, `A(i,k)` from `S2` and `A(j,l)` from `S3`, or choosing `A(k,k)` from `S1`, `A(i,k)` from `S2` and `A(j,k)` from `S3`. Since both these shackles are legal, their Cartesian product (in either order) is legal. It can be shown that one order gives a fully-blocked left-looking Cholesky, identical to the blocked Cholesky algorithm in [24], while the other order gives a fully-blocked right-looking Cholesky.

3.2.9 Shackling for multiple levels of memory

Cartesian products can be used in an interesting manner to block programs for multiple levels of memory hierarchy. The high-level idea is the following: For a multi-level memory hierarchy, a product of Cartesian products of shackles is generated, where each factor in the outer Cartesian product determines enhancing locality for one level of the memory hierarchy.

```
do t1 = 1, (n+63)/64
do t2 = 1, (n+63)/64
do t3 = 1, (n+63)/64
  do t7 = 8*t1-7, min(8*t1,(n+7)/8)
  do t8 = 8*t2-7, min(8*t2,(n+7)/8)
  do t9 = 8*t3-7, min(8*t3,(n+7)/8)
    do t13 = 8*t9-7, min(n,8*t9)
    do t14 = 8*t8-7, min(n,8*t8)
    do t15 = 8*t7-7, min(8*t7,n)
      C(t13,t14) = C(t13,t14) + A(t13,t15) * B(t15, t14)
```

Figure 3.14: Matrix multiply shackled for two levels of memory hierarchy

The first term in the outer Cartesian product corresponds to shackling for the slowest (and largest) level of the memory hierarchy and corresponds to largest block size. Subsequent terms correspond to blocking for faster (and usually smaller) levels of the memory hierarchy. Figure 3.14 demonstrates this idea for matrix multiplication. The outer Cartesian product for this example has two factors: the first factor is itself a product of two shackles (on `C(i,j)` and `A(i,k)` with block sizes of 64), and the second factor is also a product of two shackles (once again, on `C(i,j)` and `A(i,k)`, but block sizes of 8). As can be seen from the code, the first term of the outer Cartesian product performs a 64-by-64 matrix multiplication, which is broken down into several 8-by-8 matrix multiplications by the second term in this product. Choosing innermost shackles of small sizes, and unrolling the resulting loops which have a small trip count, one can enhance locality for registers.

3.3 Integration of Shackling into the SGI compiler

While Section 3.1 provides information regarding the mechanisms to get data shackling to enhance locality, there are several significant policy decisions that have not been answered. The author of this dissertation spent the summer of 1997 integrating shackling into the production compiler at Silicon Graphics. During this process, several important questions had to be answered regarding these policies. This section discusses the questions and the solutions that were developed.

There were six important questions to be addressed during this integration.

1. What is the unit of transformation for a data shackle?
2. What is the orientation of the cutting planes?
3. What is the separation of cutting planes (block sizes)?
4. What is the order of traversal of blocks?
5. How are data-centric references chosen?
6. How can localization constraints be folded into loop bounds?

One approach to answering these questions is to treat them as classical optimization problems, and try to find optimal solutions in the presence of an accurate model. However, this approach is impractical in a production compiler. Simple and effective heuristics were developed instead.

3.3.1 Unit of Transformation

The unit of transformation in the implementation currently is a single imperfectly nested loop. In principle, shackling can be applied across multiple imperfectly nested loops and even across procedure boundaries and this is being investigated currently. As of the time of writing this dissertation, the heuristics to be chosen to work across multiple imperfectly nested loops are still open.

3.3.2 Orientation of Cutting Planes

Cutting plane orientations were chosen to be always parallel to the data co-ordinate axes. "Skewed" blocks have not been considered. This was because skewed blocks are more likely to produce complicated loop bounds, which can be detrimental to subsequent phases such as software pipelining.

3.3.3 Block Sizes

To determine block sizes, a simplified version of the memory model already in use in the SGI compiler was used for shackling. The model currently in use is based on the assumption that a cache has an *effective size*, such that if the amount of data in a cache is smaller than its effective size, conflict misses are relatively unimportant. The model estimates the amount of data touched by a given loop nest(called a *footprint*), and computes tile sizes such that for loops of interest, the footprint of the tiled loop nest is smaller than the effective cache size. A simple variant of this scheme has been developed for use with data shackling.

The fundamental assumption in data shackling is that the data touched by a single execution of the imperfectly nested loop under consideration must fit in the effective cache size for a single instance of the composite shackle. This in turn requires a method to estimate parametrically the amount of data touched by a composite shackle. This is done in three steps:

1. In the first step, references responsible for most of the data touched in the imperfectly nested loop are identified. For every statement, a set is formed identifying this set of references. For a statement nested most deeply in a program, this set is defined to be all the references from that statement with the highest (row)-ranked access matrices. For example, in matrix multiplication, `A(i,k)`, `B(k,j)` and `C(i,j)` all correspond to access matrices of row-rank 2, and are all chosen in this step. In Cholesky Factorization, the three references chosen are `A(i,j)`, `A(i,k)` and `A(j,k)` — `A(k,k)` is not chosen because its access matrix has row rank 1, which `A(i,k)` is not chosen because it appears in a statement nested inside only two loops. For statements not most deeply nested in a loop nest, this set is defined to be empty.
2. In the second step, the references chosen in the previous step are partitioned into groups — two references to the same array fall into the same group if their access matrices have the same linear part, but possibly different affine parts. References belonging to two different arrays always fall into different groups. For example, in matrix multiplication, `A(i,k)`, `B(k,j)` and `C(i,j)` all fall into different groups. Similarly, in Cholesky factorization, `A(i,j)`, `A(i,k)` and `A(j,k)` all fall into different groups. On the other hand, two

references of the form `A(i,j)` and `A(i+1,j)` would be assigned to the same group. The assumption is that all references assigned to the same group enjoy perfect reuse, while references assigned to different groups enjoy no reuse. In such a situation, the footprint of the entire loop nest is simply the sum of the contributions from each of the groups of references. Finally, two groups from two different statements referring to the same array are merged if all the references in the two groups under consideration have the same linear part, and the two statements under consideration have identical data-centric references. If E represents the effective size of the cache and g represents the number of groups, then each group is allowed to have a footprint as large as E/g.

3. The last step involves computing the footprint of every group for a single instance of a composite shackle. Two simplifications are applied to this computation — (i) the footprint of a group is approximated by the footprint of a *single* reference picked at random from the group, and (ii) it is assumed that the footprint of the group is identical for all instances of the composite shackle. The first simplification is justified by the assumption that all references in a group enjoy perfect reuse, and the second simplification is justified for sufficiently large array sizes and loop bounds, where boundary effects do not dominate. The reference picked for each group is called the *representative reference*.

   Evaluating the footprint of a single reference for a single instance of the composite shackle is straightforward. Variations of this problem have been addressed in the literature [39]. A single instance of the composite shackle is completely specified by a specific set of values for the block coordinates for each level of a composite shackle - for the sake of simplicity, each of the block coordinates can be assumed to be 0. In addition, it is assumed that all blocked dimensions of an array have the same block sizes - in other words, for every array `A`, a single unknown parameter B_A denotes the block size for that array. A system of linear integer equations expressing the localization constraints corresponding to the composite shackle is assembled for the statement containing the *representative reference*. The number of distinct elements touched by the representative reference under this system of equations is multiplied by the size of each element to yield a polynomial in a single parameter for the footprint for this reference. Determining the number of distinct elements touched by the representative reference can be formulated as counting the number of integer solutions inside a convex polyhedron, and is currently estimated by counting the number of integer solutions inside the bounding box for the convex polyhedron. Sophisticated approaches such as Ehrhart Polynomials [16] can potentially be used to obtain better solutions in practice.

A detailed evaluation of this scheme for determining block sizes for the factorization codes in LAPACK is presented in Section 3.4.

3.3.4 Order of traversal of Blocks

Once an array was blocked, the order of traversal of blocks was chosen to be the simple lexicographic order on block co-ordinates. For example, for a two-dimensional array, the blocks are visited from left to right, and within a given block column, from top to bottom.

3.3.5 Choice of Data-centric references - How much to Compose?

Composing data shackles provides finer control over data accesses in the blocked code. As discussed earlier, shackling just one reference in matrix multiplication (say `C(i,j)`) does not constrain all the data accesses. On the other hand, shackling all three references in this code is over-kill since shackling any two references constraints the third automatically. Applying too many levels of composition does not affect the correctness of the code, but it introduces unnecessary loops into the resulting code which must be optimized away by the code generation process to get good code. The following obvious result is useful to determine how far to carry the process of taking Cartesian products. It is assumed that all array access functions are linear functions of loop variables (if the functions are affine, the constant terms can be dropped). In such a situation, the array access function can be written as $F * \underline{I}$ where F is the **data access matrix** [31] and $\underline{I}$ is the vector of iteration space variables of loops surrounding this data reference.

Theorem 2 *For a given statement S, let $F_1, \ldots, F_n$ be the access matrices for the* *shackled data references in this statement. Let F_{n+1} be the access matrix for an unshackled reference in S. Assume that the data accessed by the shackled references are bounded by block size parameters. Then the data accessed by F_{n+1} is bounded by block size parameters iff every row of F_{n+1} is spanned by the rows of $F_1, \ldots, F_n$.*

Stronger versions of this result can be proved, but it sufficed for purposes of the implementation. For example, the access matrix for the reference `C(i,j)` is $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$. Shackling this reference does not bound the data accessed by row $\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$ of the access matrix $\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ of reference `B(k,j)`. However, taking the Cartesian product of this shackle with the shackle obtained from `A(i,k)` constrains the data accessed by `B(k,j)`, because all rows of the corresponding access matrix are spanned by the set of rows from the access matrices of `C(i,j)` and `A(i,k)`. Composition is applied if even a single assignment statement from the loop nest under consideration stands to benefit as a result.

The first data-centric reference chosen is always the one for which the access matrix [49] has the highest rank among all the references in that statement. In the presence of multiple references with highest rank, preference is given to a reference on the left hand side of the assignment statement, but otherwise the choice is arbitrary.

```
do i = 1, 100            do i = 1, 100
  S0(i)                    S0(i)
  do j = 1, 200            do j = 1, 200
    if (i <= 50)             if (i <= j)
      S1(i,j)                  S1(i,j)
    S2(i,j)                  S2(i,j)
  S3(i)                    S3(i)
```

(a) (b)

Figure 3.15: Two instances of folding a conditional into a loop bound

3.3.6 Folding Localization constraints into loop bounds

As has already been noted, the naive code generated by the specification of a data shackle consists of newly introduced affine conditionals surrounding assignment statements. The role of these affine conditionals, known as *localization constraints*, is to restrict the amount of data touched by the data-centric reference. However, since the localization constraints are introduced inside innermost loops, the naive code generated by the specification of the shackle will have poor performance unless these conditionals are optimized away. This optimization is accomplished by "folding" the conditionals into loop bounds. This section discusses this problem in the general context of optimizing away affine conditionals in a loop nest by folding them away into loop bounds.

The examples from Figure 3.15 (a) & (b) demonstrate the two possibilities that arise when a conditional is to be folded into a loop. Either the conditional is invariant in the innermost loop containing the conditional (as in Figure 3.15(a)), or the conditional is not invariant in the innermost loop containing it (as in Figure 3.15(b)).

It is obvious that when the affine conditional to be eliminated is invariant inside the innermost loop surrounding it, it is wasteful to evaluate the conditional inside the innermost loop, and the evaluation can in fact be hoisted out. Conceptually, this corresponds to transforming the code in Figure 3.16(a) to that in Figure 3.16(b). The `j` loop is duplicated, and each copy of the `j` loop is surrounded by a conditional derived from the loop invariant conditional surrounding `S1` — one instance of the `j` loop is surrounded by the same conditional as surrounds `S1`, the other instance of the `j` loop is surrounded by the negation of the conditional surrounding `S1`. This code can next be identically simplified to the code in Figure 3.16(c) by recognizing that after the previous transformation, the conditional surrounding `S2` is guaranteed to be true, while the conditional surrounding `S2` in the second of the copy of the `j` loop is guaranteed to be false. In other words, the code in Figure 3.15(a) can be transformed to that in Figure 3.16(d).

After a conditional has been hoisted around all loops with respect to which it is invariant, eventually a state is reached when the conditional is no longer invariant with respect to the loop surrounding it. This is the case with the conditional surrounding `S2` in Figure 3.15(b) and the conditionals surrounding the `j` loop in Figure 3.16(d). The key to handling the conditional in this case is under certain conditions, the range of the surrounding loop can be split into two ranges, such the conditional is guaranteed to be true in one of the ranges and false in the other. For example, when the conditional `i`$\leq$ `50` is considered in Figure 3.16(d), the range of the `i` loop can be divided into the ranges $[1 \ldots 50]$ and $[51 \ldots 100]$. `i`$\leq$ `50` is always true when `i` is in the range $[1 \ldots 50]$ and always false when `i` is in the range

```
do i
  do j
    if (F(i) S1
    S2
```

(a)

```
do i
  if (F(i))
    do j
      if (F(i) S1
      S2
  if (!F(i))
    do j
      if (F(i)) S1
      S2
```

(b)

```
do i
  if (F(i))
    do j
      S1
      S2
  if (!F(i))
    do j
      S2
```

(c)

```
do i = 1, 100
  S0(i)
  if (i ≤ 50)
    do j = 1, 200
      S1(i,j)
      S2(i,j)
  if (i ≥ 51)
    do j = 1, 200
      S2(i,j)
  S3(i)
```

(d)

Figure 3.16: Dealing with a conditional invariant w.r.t the loop surrounding it

[51 . . . 100]. This produces the code in Figure 3.17(a), which can be identically simplified to the code in Figure 3.17(b). The key point to note is that the point where the range of the loop index is split must be guaranteed to be within the bounds of the split loop for this transformation to be legal — in this example, 50 is guaranteed to be within 1 and 100, making the transformation legal.

Figure 3.15(b) represents a more subtle case where the conditional is not invariant with respect to the innermost loop. In this case, the range of the `j` loop can be

```
do i = 1, 50
  S0(i)
  if (i ≤ 50)
    do j = 1, 200
      S1(i,j)
      S2(i,j)
  if (i ≥ 51)
    do j = 1, 200
      S2(i,j)
  S3(i)
do i = 51, 100
  S0(i)
  if (i ≤ 50)
    do j = 1, 200
      S1(i,j)
      S2(i,j)
  if (i ≥ 51)
    do j = 1, 200
      S2(i,j)
  S3(i)
```

(a)

```
do i = 1, 50
  S0(i)
  do j = 1, 200
    S1(i,j)
    S2(i,j)
  if (i ≥ 51)
    do j = 1, 200
      S2(i,j)
  S3(i)
do i = 51, 100
  S0(i)
  if (i ≥ 51)
    do j = 1, 200
      S2(i,j)
  S3(i)
```

(b)

Figure 3.17: Splitting the range of a loop to eliminate a conditional - Simple Case

split into the ranges $[1 \ldots \mathtt{i}-1]$ and $[\mathtt{i} \ldots 200]$. In this case, the split point for the `j` loop is not at a constant point, but depends on the value of the loop index variable of the surrounding loop(`i`). However, the transformation splitting the range of the `j` loop is still guaranteed to be legal, because for all values of `i`, it can be proved that `i` lies within 1 and 200.

These observations form the basis of the implementation of optimizing affine conditionals by folding them into loop bounds. Hereafter, the optimization of affine conditionals by folding into loop bounds is called *If simplification*. For the sake

```
do i = 1, 100                 do i = 1, 3
  Affine Conditional            S1
  if (i <= 3)                   S2
    S1                        do i = 4, 100
  S2                            S2
```

(a) (b)

Figure 3.18: Folding an affine conditional into a loop bound - (a) before, (b) after

of simplicity, it is assumed that every test in a conditional corresponds to a single affine constraint. A test that corresponds to a conjunction of affine constraints can be represented as a series of nested conditionals, where the test for each conditional corresponds to a single affine constraint. Disjunctions of affine constraints are not handled.

Every affine constraint can be normalized to the representation $\sum_{i=1}^{n} a_i * x_i \leq c$, where a_i and c are integers and x_i corresponds to either a loop index variable or a constant parameter that appears in the affine constraint.

Definition 6 *The normalized representation of the test of a conditional testing for an affine constraint is defined to be the* access vector *for the conditional.*

The following convention is observed regarding normalization: If a conditional is nested inside l loops and its test involves p constant parameters including the loop index variables, then the access vector for the conditional is represented as $\sum_{i=1}^{l+p} a_i * x_i \leq c$, where $x_1 \ldots x_p$ represent the constant parameters in the program and $x_{p+1} \ldots x_{l+p}$ represent the loop index variables for the loops surrounding this conditional. Using this convention, the access vector for the conditional in Figure 3.18 is $x_1 \leq 3$.

Definition 7 *The* ith enclosing do loop *of a conditional is defined as the ith do loop encountered while following the parents of the conditional till the root of the enclosing function is encountered.*

For the conditional in Figure 3.18, the `i` loop is the first enclosing do loop. There are no *ith* enclosing do loops for this conditional for $i > 1$.

Definition 8 *Let A be the access vector for a conditional C nested inside l loops and involving p constant parameters. The* sinkable index for the conditional *for this conditional is defined as follows:*

- 0 *if* $a_{p+1} \ldots a_{l+p}$ *are all zeros*
- $(\max_i a_i \neq 0) - p$ *otherwise.*

If the sinkable index i for the conditional is nonzero, the potentially sinkable do loop *for the conditional is defined as the ith do loop enclosing the conditional.*

For the affine conditional in Figure 3.18(a), the sinkable index is 1 and the potentially sinkable do loop is the `i` loop.

Definition 9 *Let s be the sinkable index for a conditional C. In the access vector for C, it must be the case that $a_s \neq 0$, and $\forall i$ such that $s < i \leq l + p$, $a_i = 0$. The* discriminant *for this conditional, denoted by $\mathfrak{D}(C)$ is defined as $c - \sum_{i=1}^{s-1} a_i * x_i$.*

For the affine conditional in Figure 3.18, the discriminant is simply equal to 3.

Definition 10 *Let s be the sinkable index for a conditional C. The upper bound induced by this conditional, denoted by $\mathfrak{UB}_{if}(C)$ is defined as follows:*

$$\begin{cases} \lfloor \frac{\mathfrak{D}(C)}{a_s} \rfloor & a_s > 0 \\ \lfloor \frac{-\mathfrak{D}(C)-1}{a_s} \rfloor & a_s < 0 \end{cases} \tag{3.3}$$

The conditional naturally imposes a lower bound *if $a_s < 0$, and* naturally imposes an upper bound *otherwise.*

The upper bound induced by the affine conditional in Figure 3.18 is equal to 3. Also, the conditional naturally imposes an upper bound.

Definition 11 *The lower bound induced by an affine conditional C, denoted by $\mathfrak{LB}(C)_{if}$ is defined to be $\mathfrak{UB}_{if}(C) + 1$.*

The lower bound induced by the affine conditional in Figure 3.18 is equal to 4.

The following theorem is stated without proof.

Theorem 3 *Given an affine conditional C, let l be the potentially sinkable do loop for C. Let l_i represent the corresponding loop index variables. Then, given the two intervals $(-\infty, \mathfrak{UB}_{if}(C)]$ and $[\mathfrak{LB}_{if}(C), \infty)$, C is always true when l_i takes values in one of these intervals and C is always false when l_i takes values in the other interval.*

Proof: Obvious, hence omitted. □

For example, the affine conditional in Figure 3.18 is always true when `i` is in the range $(-\infty, 3]$ and always false when `i` is in the range $[4, \infty)$.

```
        ⋮                      ⋮
                               do l = lb, exp
        do l = lb, ub            body(l)
          body(l)              do l = exp+1, ub
        ⋮                        body(l)
                               ⋮

           (a)                        (b)
```

Figure 3.19: Index-set split of a do loop l by at expression exp: (a) before, (b) after

Definition 12 *Let C be an affine conditional and l be the potentially sinkable do loop for C. Then $\mathfrak{LB}_{do}(C)$ and $\mathfrak{UB}_{do}(C)$ are respectively defined to be equal to the upper and lower bounds of do loop l.*

The following three definitions introduce two important code transformations that are used in *If Simplification* - Index set splitting and Promotion.

Definition 13 Index-set splitting *of a loop l by an expression e is defined as follows - let lb and ub be the lower and upper bounds of l respectively. Then index-set splitting this loop by exp creates two instances of l. In the first instance of l, the loop index variable ranges from lb through exp. In the second instance of l, the loop index variable ranges from $exp + 1$ through ub. This transformation is pictorially depicted in Figure 3.19. Note that this transformation is legal iff $lb \leq exp \leq ub$.*

Definition 14 *The inversion of an affine test* `x` *is denoted as* `!x`*.* `!x` *is false whenever* `x` *is true and is true whenever* `x` *is false.*

Definition 15 Promoting *a loop-invariant affine test* `x` *around a loop l is defined*

```
                      ⋮
                      if (x) then
    ⋮                   do l = lb, ub
    do l = lb, ub         body(l)
      body(l)         if (!x) then
    ⋮                   do l = lb, ub
                          body(l)
                      ⋮

         (a)                 (b)
```

Figure 3.20: Promoting an affine test `x` around a loop l: (a) before, (b) after

as follows: Two instances of l are created. A conditional is introduced around the first instance of l with the test enforcing that `x` *is true. A second conditional is introduced around the second instance of l testing that* `!x` *is true. Again, it should be noted that* `x` *should be invariant in l, i.e. x can only involve an index variable if it is derived from a loop that is an ancestor of l, or constant parameters. Figure 3.20 pictorially depicts this transformation. It can be proved in a straightforward manner that this transformation is always legal.*

Given an affine conditional C which should be sunk into a loop, the first step is to determine the four quantities $\mathfrak{LB}_{if}(C)$, $\mathfrak{UB}_{if}(C)$, $\mathfrak{LB}_{do}(C)$ and $\mathfrak{UB}_{do}(C)$. In the following discussion, let l refer to the potentially sinkable do loop for C. Once these quantities are available, C can be dealt with according to which of the following four cases apply:

1. The conditional can be proved to be trivially false. A simple linear integer system of equalities and inequalities can be formed to determine this. If this is

the case, the conditional and all statements contained inside it can be deleted.

2. The conditional can be proved to be trivially true. Again, a simple linear integer system of equalities and inequalities can be formed to determine this. If this is the case, the conditional itself can be deleted, however, its body is left unchanged.

3. Either $\mathfrak{LB}_{if}(C)$ or $\mathfrak{UB}_{if}(C)$ can be proved to strictly within the range $[\mathfrak{LB}_{do}(C)$, $\mathfrak{LB}_{do}(C)]$. Again, this can be determined by solving a system of integer linear constraints. If this is the case, index set splitting is applied to loop l at index value $\mathfrak{LB}_{if}(C)$. This essentially creates two instances of l, one with its loop index value in the range $[\mathfrak{LB}_{do}(C), \mathfrak{UB}_{if}(C)]$ and the other with its loop index variable ranging through $[\mathfrak{LB}_{if}(C), \mathfrak{UB}_{do}(C)]$

4. If none of the above three cases apply, then two affine tests are derived from C and promoted around l. If C naturally imposes a lower bound, the two conditions promoted around l are $(\mathfrak{LB}_{if}(C) \leq \mathfrak{LB}_{do}(C))$ and $(\mathfrak{LB}_{if}(C) \leq \mathfrak{UB}_{do}(C))$. If C naturally imposes an upper bound, the two conditions promoted around l are $(\mathfrak{UB}_{if}(C) \geq \mathfrak{UB}_{do}(C))$ and $(\mathfrak{UB}_{if}(C) \geq \mathfrak{LB}_{do}(C))$.

Folding conditionals into loop bounds is then an iterative process. From a program, a conditional is picked which can be folded into a loop bound. This process may or may not introduce additional conditionals. After the code transformations related to the present conditional have been dealt with, another conditional is picked and the whole process is repeated. A simple inductive argument can be used to prove that this process terminates.

3.4 Evaluation of Shackling on LAPACK kernels

This section discusses the performance of the code produced by shackling the matrix factorization codes found in LAPACK.

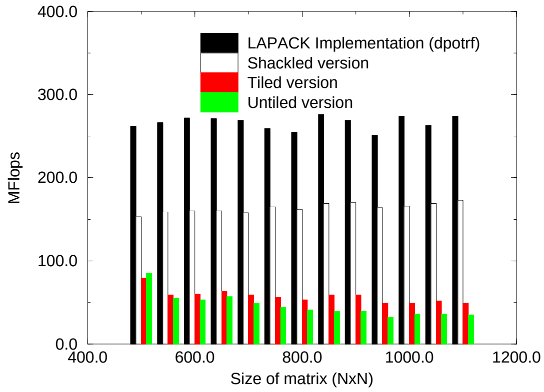
3.4.1 Cholesky factorization

The shackled code produced by the compiler was generated by composing two shackles. In both shackles, the array was divided into square blocks (the compiler heuristic chose 70x70 blocks), and these blocks were visited in left-to-right, top-to-bottom order. In the outer shackle, the compiler chose the left-hand side reference from each assignment statement for shackling, while in the inner shackle, the compiler selected a reference from the right-hand side of each statement: `A(k,k)` for the square root statement, and `A(i,k)` for the scale and update statements. This code performs better than the tiled code in both versions of `kij` Cholesky because it can exploit reuse at all levels. The shackled code is also impervious to the change in the input code.

The same shackle was used for all other versions of Cholesky factorization as well. The high-level structure of the code in every case is identical, but the performance of the different shackled versions in Figures 3.21:3.26 varies because intra-block iterations are done in source-program order.

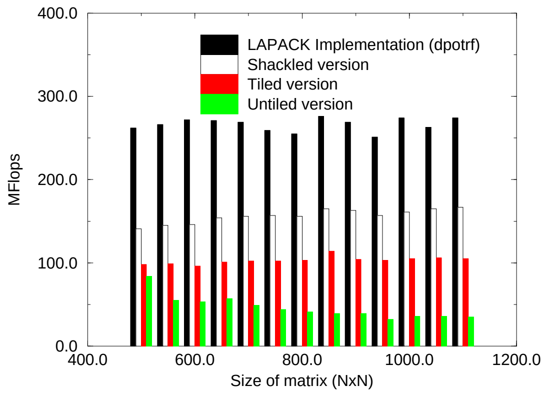
Permuting the two update loops in the `kij` version yields the `kji` version. This version is not an SNL, and hence tiling is ineffective. However, legal shackled code can be obtained using the same two-level composite shackle as in the case of both

```
do k = 1, NMAX
  A(k,k) = dsqrt (A(k,k))
  do i = k+1, NMAX
    A(i,k) = A(i,k) / A(k,k)
  do i = k+1, NMAX
    do j = k+1, i
      A(i,j) -= A(i,k) * A(j,k)
```



(a) Cholesky factorization: Version kij (distributed)

```
do k = 1, NMAX
  A(k,k) = dsqrt (A(k,k))
  do i = k+1, NMAX
    A(i,k) = A(i,k) / A(k,k)
    do j = k+1, i
      A(i,j) -= A(i,k) * A(j,k)
```



(b) Cholesky factorization: Version kij (fused)

Figure 3.21: Cholesky Factorization: kij version

```
do k = 1, NMAX
  A(k,k) = dsqrt (A(k,k))
  do i = k+1, NMAX
    A(i,k) = A(i,k) / A(k,k)
  do j = k+1, NMAX
    do i = j, NMAX
      A(i,j) -= A(i,k) * A(j,k)
```

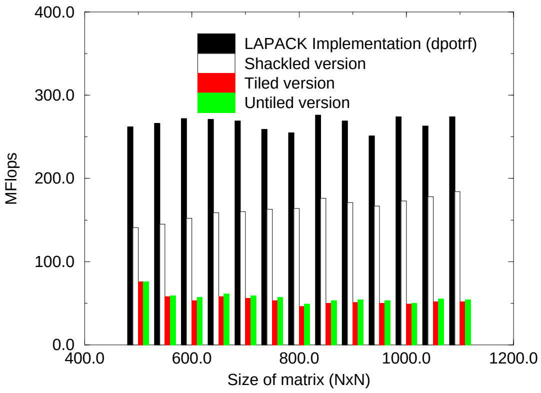


Figure 3.22: Cholesky Factorization: kji version

the `kij` versions. The array `A` was divided into 70×70 blocks as in the previous case. The shackled code again significantly outperforms the tiled code; because of spatial locality in the update loops, the shackled code in this case performs somewhat better than for either of the `kij` versions.

The versions of Cholesky factorization examined so far are right-looking in the sense that they perform update *eagerly*. Performing the updates *lazily* yields several more versions. Figure 3.23 shows the `jik` version of Cholesky factorization and compares the performance of shackling and tiling on this version. As mentioned before, tiling can accomplish little; however the work in the update loops is essentially a matrix-vector product, which is optimized very well by the SGI compiler by using outer loop unrolling and scalar replacement to reduce the number of store instructions. This causes the tiled and untiled versions to perform very well for small problem sizes. Shackling, as in the case of other versions, exploits reuse for all loops and outperforms the tiled version by almost a factor of 2 as problem size

```
do j = 1, NMAX
  do i = j, NMAX
    do k = 1, j-1
      A(i,j) -= A(i,k) * A(j,k)
   A(j,j) = dsqrt (A(j,j))
   do i = j+1, NMAX
     A(i,j) = A(i,j) / A(j,j)
```

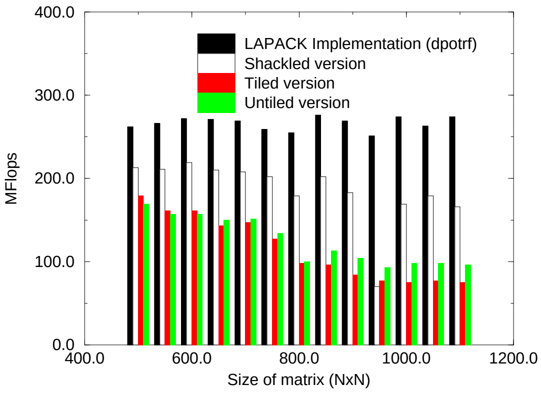


Figure 3.23: Cholesky Factorization: Version jik

increases. The shackled code has a performance dip for a problem size of 950 because of a sudden increase in the number of conflict misses. This is because the memory model for shackling does not factor in conflict misses when estimating the block sizes. The `jki` version is obtained from the `jik` version by permuting the `i` and `k` loops. The behavior of the shackled code for both the versions is identical.

The final two versions of Cholesky are the `ijk` and `ikj` versions. In both these versions, shackling performs worse than for the previous versions because of poor interaction with the register optimizer. However, it still significantly outperforms tiling in both these cases (by almost a factor of 5 for the `ijk` version and a factor of 2 for the `ikj` version.

Finally, Figure 3.27:Figure 3.33 show the breakdown of performance from shackling and tiling by separately identifying the contributions from registers and the cache. Three points are worth noting:

1. Tiling for the cache has significant improvement only in one version (the fused

```
do j = 1, NMAX
  do k = 1, j-1
    do i = j, NMAX
      A(i,j) -= A(i,k) * A(j,k)
  A(j,j) = dsqrt (A(j,j))
  do i = j+1, NMAX
    A(i,j) = A(i,j) / A(j,j)
```

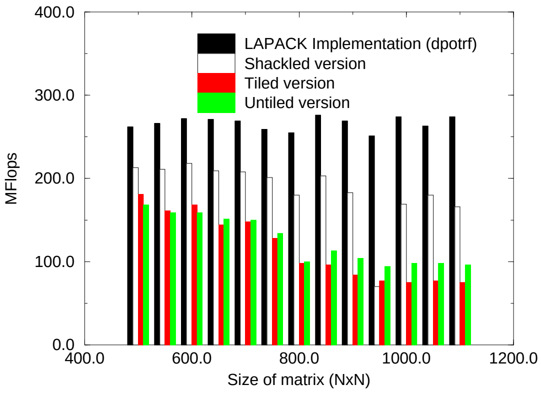


Figure 3.24: Cholesky Factorization: Version jki

variant of `kji` version).

2. Shackling for L2 cache results in significant benefit in all cases. In addition, with the exception of the `ijk` version, shackling results in significant improvement from the register tiling routine as well.

3. In all cases, the benefit from register tiling of the shackled code always exceeds the benefit from register tiling obtained without shackling. In other words, the output of the shackled code interacts very favorably with the built-in register tiling routine in the SGI compiler.

3.4.2 LU factorization

Figure 3.34 compares the performance of shackling and tiling for LU factorization with pivoting. As mentioned before, the entire loop nest is not an SNL, and therefore cannot be tiled. However, the update loop nest can be tiled, and this has a small benefit because it permits spatial locality to be exploited.

```
do i = 1, NMAX
  do j = 1, i-1
    do k = 1, j-1
      A(i,j) -= A(i,k) * A(j,k)
    A(i,j) = A(i,j)/A(j,j)
  do k = 1, i-1
      A(i,i) -= A(i,k) * A(i,k)
  A(i,i) = dsqrt (A(i,i))
```

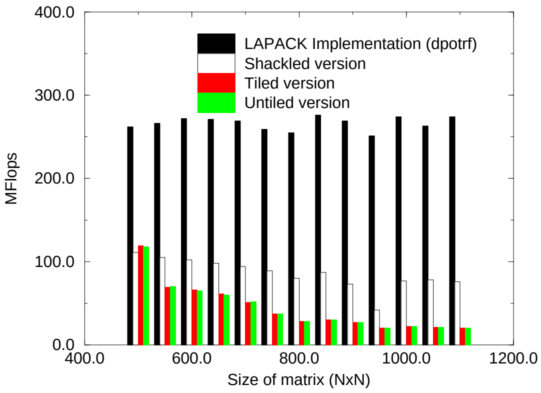


Figure 3.25: Cholesky Factorization: Version ijk

Shackling the entire factorization code requires mechanisms to deal with (i) scalars, (ii) conditionals involving non-loop index variables and (iii) the case when the array to be shackled does not appear in all statements. These issues are all addressed next:

The first key idea is that when an array is being shackled, the data-centric reference being chosen for a statement can actually occur in a different statement. In Figure 3.35, let `A` be shackled. This poses no problems for `S2` and `S3`, each of which contains a reference to `A` that can be shackled. `S1` however contains no references to `A`. To apply data shackling to the entire program however, a data-centric reference must be picked for `S1` as well. This is done in two steps. The first step is to allow a reference to `A` occurring somewhere in the program other than in `S1` to be picked as a data-centric reference for `S1`. For example, if `A(2*i,2*j)` is picked as the data-centric reference for `S1`, then the same localization constraints are introduced around `S1` as for `S2`. The second step is to realize that not every reference to `A` in

```
do i = 1, NMAX
  do k = 1, i-1
    A(i,k) = A(i,k) / A(k,k)
    do j = k+1, i
      A(i,j) -= A(i,k) * A(j,k)
  A(i,i) = dsqrt (A(i,i))
```

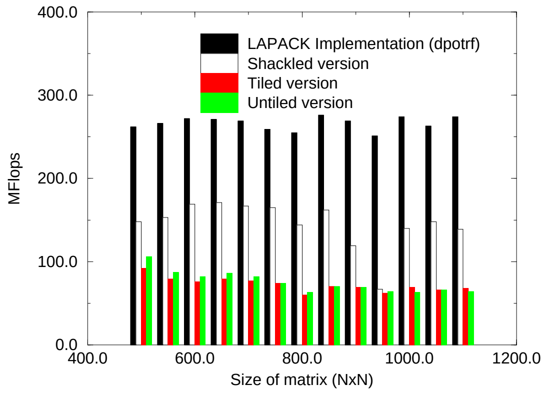
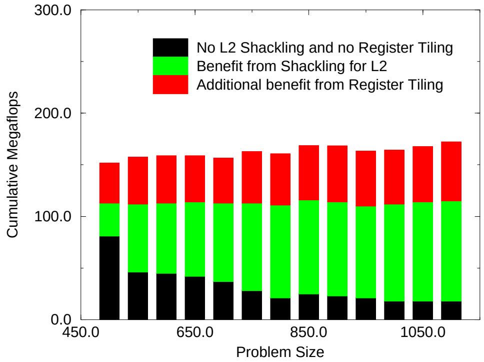


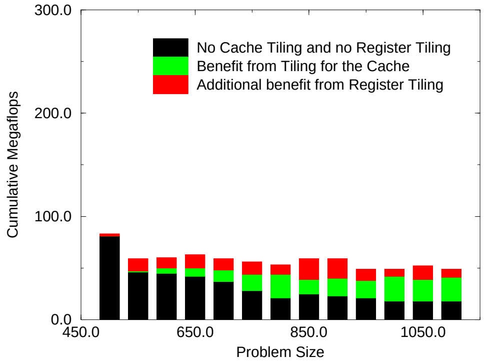
Figure 3.26: Cholesky Factorization: Version ikj

the program need be considered as a potential data-centric reference for `S1`. The loop independent flow dependence from `S1` to `S2` caused by scalar `s` implies that each assignment to `s` must have its value consumed before being assigned again. In other words, dynamic instances of `S1` and `S2` corresponding to the same `i` and `j` values must be executed one after the other, with no intervening executions of either `S1` or `S2`. This is guaranteed by choosing the same data-centric reference for `S1` and `S2`. In other words, `A(2*i,2*j)` from `S2` must be made available to `S1` as a candidate data-centric reference.

How can it be determined that `A(2*i,2*j)` should be made available to `S1` as a candidate data-centric reference? One choice is to make *every* reference to `A` in the entire program available to every statement not containing any references to `A` as candidate data-centric references. However, this is wasteful. For example, in Figure 3.35, making `A(2*i,2*j+1)` will not result in a legal shackle, because this choice would violate the flow dependence on `s` from dynamic instances of `S1` to



Shackled Code

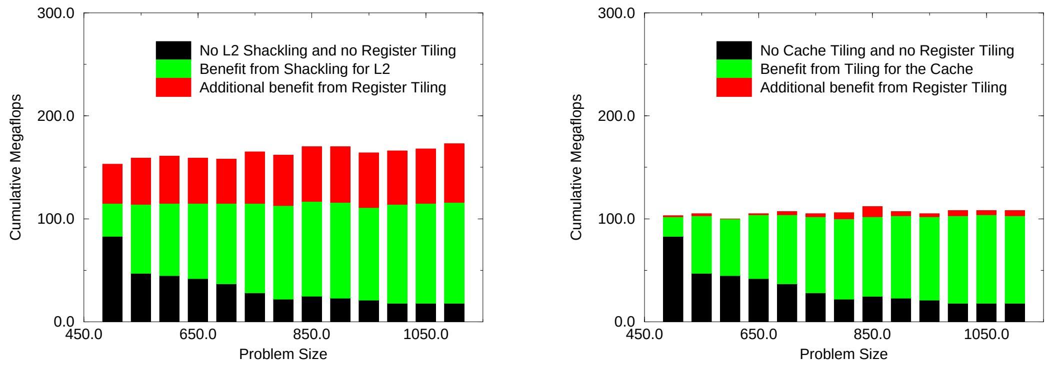


Tiled Code

Figure 3.27: Cholesky Performance Breakdown: Version kij (distributed)

dynamic instances of `S2`. Intuitively, the flow dependence from `S1` to `S2` implies that choosing the same data-centric reference for `S1` and `S2` can be useful for satisfying that dependence. However, since scalar `s` is not used at all in `S3`, there is no obvious benefit available from choosing `A(2*i,2*j+1)` the data-centric reference for `S1`, and the flow dependence from `S1` to `S2` on `s` will be violated as a result.

This notion is formalized in the algorithm presented in Figure 3.36. The key idea is that for every pair of statements `S1` and `S2`, if there is a scalar dependence from `S1` to `S2`, both `S1` and `S2` should have the same set of candidate data-centric references available. The set of candidate references for every statement is initialized to the references contained in that statement. In every iteration of the while loop in Figure 3.36, a statement is picked to see if either it uses or defined a scalar. If this statement defines (respectively uses) a scalar, then the candidate data-centric references available to this statement are added to all statements which use (respectively define) the scalar under consideration. Any statement which has its set of candidate



Shackled Code　　　　　Tiled Code

Figure 3.28: Cholesky Factorization Performance Breakdown: Version kij(fused)

references changed as a result is added back to the work queue. The procedure terminates when a fixed point has been reached, i.e. when the worklist is empty. An invocation of Procedure FindCandidates with a statement `I` identifies all statements which must have the same set of candidate references as `I` due to the presence of scalar dependences, and marks these statements as having been visited. Repeated invocations to this procedure with new unvisited statements partitions the statements in the program into equivalence classes, where each equivalence class contains statements having identical sets of candidate data-centric references.

For example, if Procedure FindCandidates is invoked initially with `S1` for the program in Figure 3.35, `S1` and `S2` are marked as visited and belong to the same equivalence class. A subsequent call to with `S3` places `S3` in its own equivalence class. Alternatively, an initial invocation with `S3` places `S3` it its own equivalence class, leaving `S1` and `S2` as unvisited. A subsequent call with either of these two statements as the argument terminates after placing both `S1` and `S2` in the same

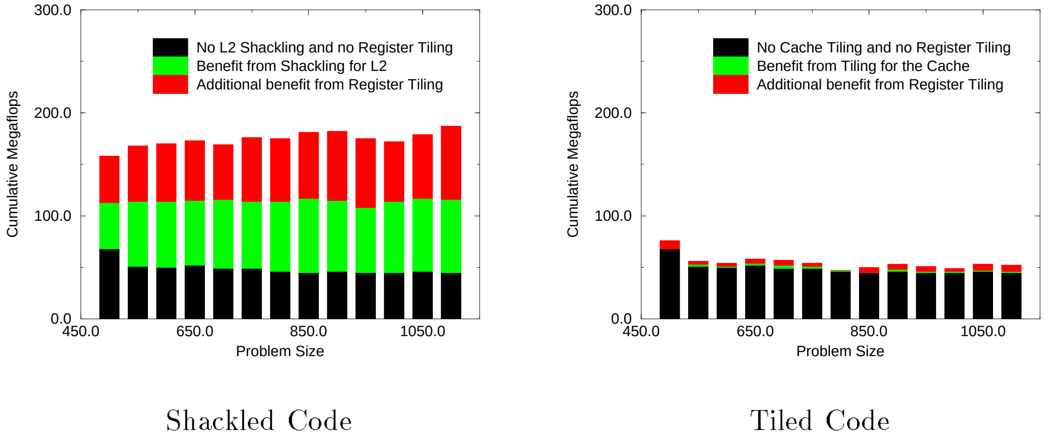
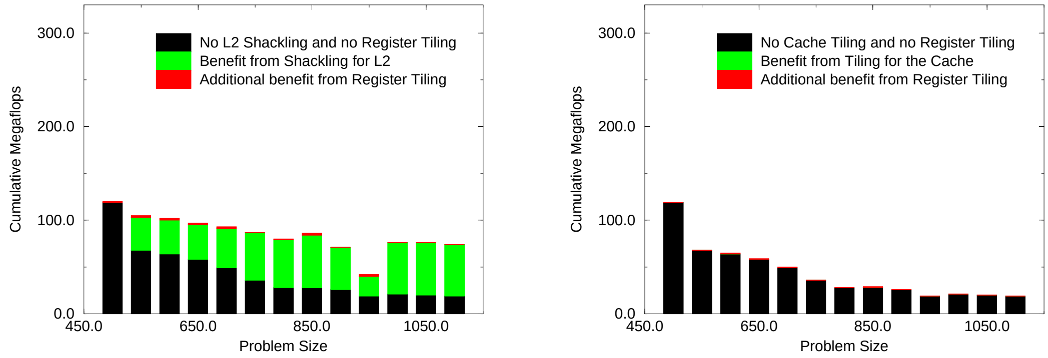


Figure 3.29: Cholesky Factorization Performance Breakdown: Version kji

equivalence class. At this time, `S1` and `S2` have the same set of candidate data-centric references for `A`, i.e. `A(2*i,2*j)`. `S3` has `A(2*i,2*j+1)` as its candidate data-centric reference.

Figure 3.37 (a) & (b) represent a more complex example. Repeated calls to FindCandidates for these two programs terminate as follows: (i) The statements in Figure 3.37(a) are partitioned into two equivalence classes, `S1` and `S3` belong in one class, and `S2` and `S4` belong in another. (ii) All the four statements in Figure 3.37(b) belong in the same equivalence class, and have the same set of data-centric references available as potential candidates (the set containing `A(i,j)` and `A(i,j+1)`. Finally, in all versions of Cholesky Factorization, the square root step, the scale step and the update step do not contain any scalars at all, and each belong to a separate equivalence class.

All statements nested inside a conditional involving non loop-index variables are control dependent on the conditional. Hence, they are treated as if each of these

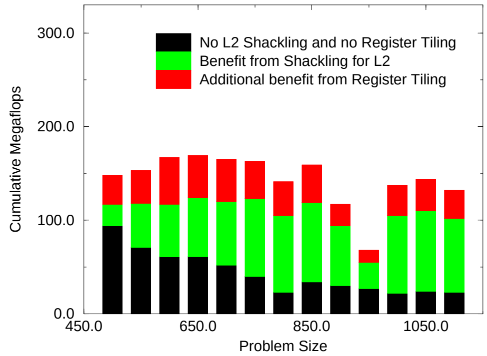


Shackled Code　　　　Tiled Code

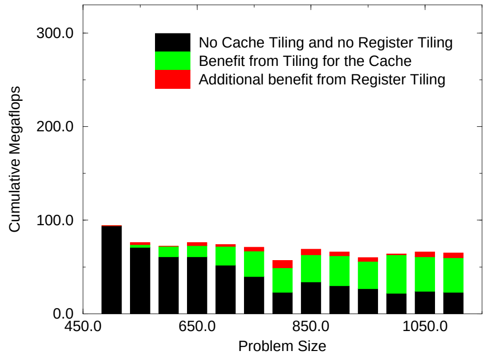
Figure 3.30: Cholesky Factorization Performance Breakdown: Version ijk

statements contain a use to the scalar involved in the test of the conditional. As a result, all statements nested inside such a conditional have the same set of candidate data-centric references available.

Using simple data-flow analysis, it can be determined automatically for the LU factorization code in Figure 3.34 that the scalar `m` needs to be expanded. The data shackle chosen by the compiler divides array `A` into block columns with block sizes ranging from 10 to 25 depending on the size of the problem. For the scale and update statements, the shackling references are chosen to be `A(i,k)` and `A(j,l)` respectively. For the three statements implementing the row permutations, the shackling references are `A(k,j)`, `A(k,j)` and `A(ipvt(k),j)` respectively, and for all the other statements, the shackling reference is `A(i,k)`. We finally note that in this particular example, the expansion of `m` can be completely free, since `ipvt(k)` represents precisely a scalar expanded `m`; however this analysis is not currently implemented.



Shackled Code



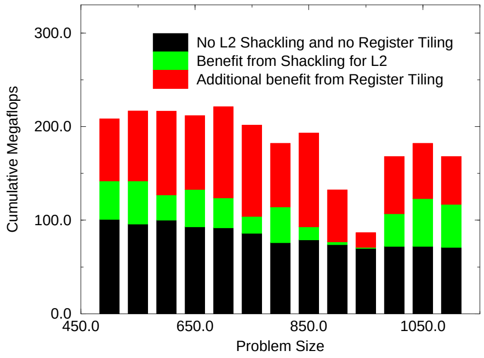
Tiled Code

Figure 3.31: Cholesky Factorization Performance Breakdown: Version ikj

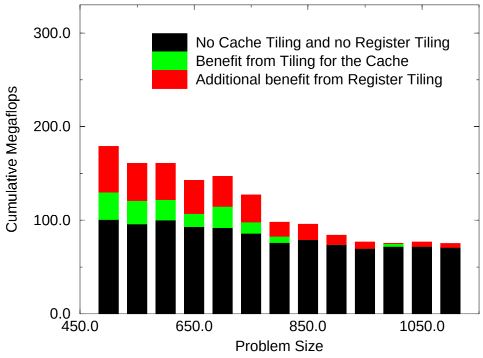
While the performance of the shackled code beats the performance of the tiled code, it is still slower than the LAPACK version by a factor of 2. This is because the LAPACK code uses domain-specific information about the commutativity of permutations and row-updates; this permits it in essence to use two-dimensional blocks rather than block columns, which results in better code. An interesting open question is how to write LU factorization with partial pivoting so that a compiler can determine this information automatically.

3.4.3 QR factorization

QR factorization performs orthogonal factorization of a matrix `A` into the product `QR` where `Q` is an orthonormal matrix and `R` is upper triangular. It is a key kernel in eigenvalue calculations. Figure 3.39 compares the performance of shackling and tiling on QR factorization using Householder reflections [24]. As in the case of LU



Shackled Code



Tiled Code

Figure 3.32: Cholesky Factorization Performance Breakdown: Version jik

factorization with partial pivoting, the array `A` is partitioned into block columns because a two-dimensional blocking is not legal. QR is similar to LU factorization except that in this case, *array expansion* of the vector `x` is required for legality. The necessary array expansion has not yet been implemented, so we modified the standard code for QR factorization to perform array expansion. Figure 3.39 shows this program. The need to expand `x` raises an important profitability question - scalar expansion is usually quite cheap, however expanding `x` creates an array as large as `A` in this case. Although shackling once again outperforms tiling, the performance of the shackled code is a factor of 2 worse than that of the LAPACK code because the LAPACK code uses domain-specific information about the associativity of matrix products to improve efficiency.

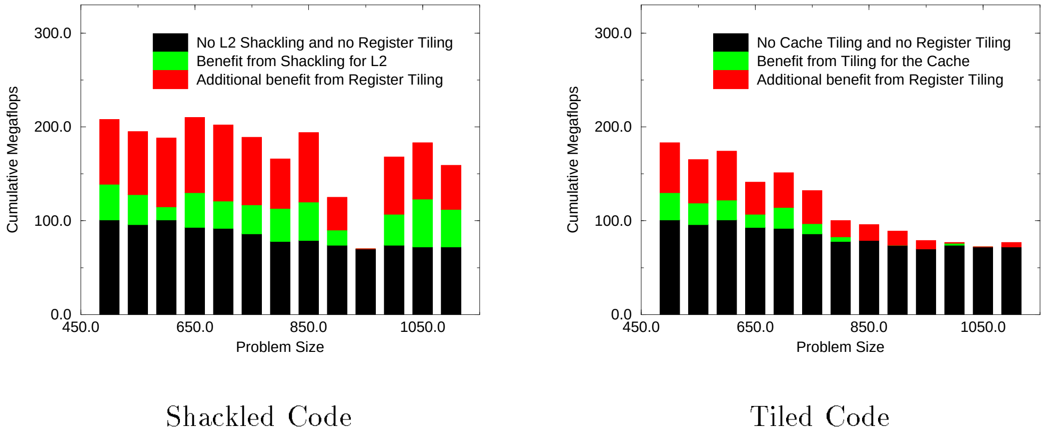


Figure 3.33: Cholesky Factorization Performance Breakdown: Version jki

3.4.4 Variation of Performance with Block Size

A final question is regarding the choice of block sizes - how good are they? There are two dimensions to this question - (i) how the shackles for the chosen block sizes perform relative to tiling, and (ii) whether these block sizes are optimal for the data-centric references chosen for the shackles. The results from the previous section show that certainly the block sizes chosen for shackling enable shackling to significantly outperform tiling for all codes. Figures 3.40:3.46 show the variation in performance of the shackled code with different block sizes. In each of these cases, the shackling phase in the compiler chooses two-dimensional blocks of size 70. The figures show the performance of the shackled code for a variety of block sizes ranging from 10 to 110. It can be seen that for five of the Cholesky versions, (`ijk`, `ikj`, `jik`, `jki` and `kji`), the block sizes chosen in the shackling implementation are very close to optimal as the problem size is increased. For the two remaining versions,

```
do k = 1, n
  temp = 0.0d0
  m = k
  //find pivot row
  do i = k, n
    d = A(i,k)
    if (ABS (d) .gt. temp)
      temp = abs(d)
      m = i
  if (m .ne. k)
    ipvt(k) = m
    //row permutation
    do j = k, n
      temp = A(k,j)
      A(k,j) = A(ipvt(k),j)
      A(ipvt(k),j) = temp
  //scale loop
  do i = k+1, n
    A(i,k) = A(i,k) / A(k,k)
  //update loops
  do i = k+1, n
    do j = k+1, n
      A(i,j) -= A(i,k) * A(k,j)
```

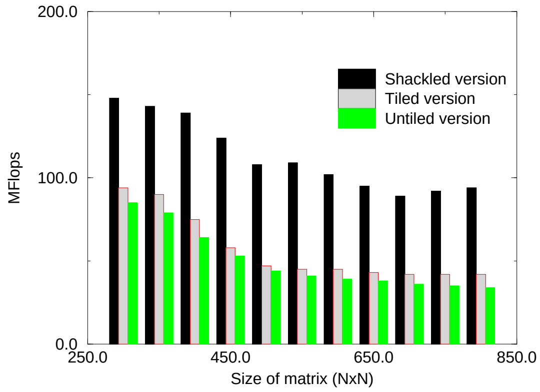


Figure 3.34: LU Factorization with Partial Pivoting - kij version

```
do i = 1, n
  do j = 1, n
    S1: s = i * j + random()
    S2: A(2*i,2*j) = s
    S3: A(2*i,2*j+1) = b(i,j+1)
```

Figure 3.35: Simple instance where an array is not in every statement

```
Procedure FindCandidates(I)
Determine candidate references for all statements
Q is a worklist of statements being processed
and is initialized to contain I
if (I has been visited)
  return;
else
  Q->Insert (I);
while (!Q->Is_Empty())
  S = Q->Get_Head();
  Mark S as visited;
  ∀ uses u in S
    ∀ definitions d reaching u
      R is the statement corresponding to d
      if (𝕮(R) ⊈ 𝕮(S))
        𝕮(R) ∪= 𝕮(S)
        Q->Insert (R);

  ∀ definitions d by S
    ∀ uses u reached by d
      R is the statement corresponding to u
      if (𝕮(R) ⊈ 𝕮(S))
        𝕮(R) ∪= 𝕮(S)
        Q->Insert (R);
```

Figure 3.36: Procedure to find the candidate data-centric references for statements

```
do i = 1, n
  do j = 1, m
    S1: s = 0.0
    S2: r = 9.1 + j
    S3: A(i,2*j) += s
    S4: A(i,2*j+1) += r
```

```
do i = 1, n
  do j = 1, m
    S1: s = 0.0
    S2: t = s + 3.0
    S3: a(i,j) += s * t
    S4: r += s * a(i,j+1)
```

Figure 3.37: Complex programs involving scalars

```
do k = 1, n
  temp = 0.0d0
  m = k
  //find pivot row
  do i = k, n
    d = A(i,k)
    if (ABS (d) .gt. temp)
      temp = abs(d)
      m = i
  if (m .ne. k)
    ipvt(k) = m
    //row permutation
    do j = k, n
      temp = A(k,j)
      A(k,j) = A(ipvt(k),j)
      A(ipvt(k),j) = temp
  //scale loop
  do i = k+1, n
    A(i,k) = A(i,k) / A(k,k)
  //update loops
  do j = k+1, n
    do i = k+1, n
      A(i,j) -= A(i,k) * A(k,j)
```

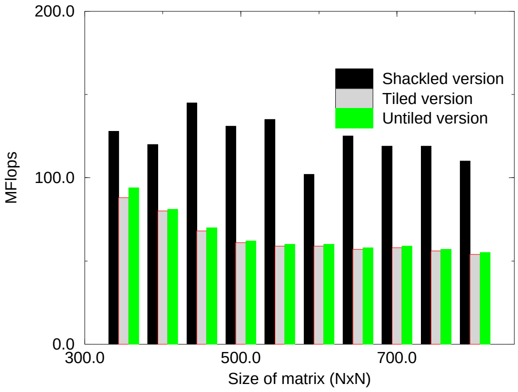


Figure 3.38: LU Factorization with Partial Pivoting - kji version

the chosen block size performs quite well, but a significantly smaller block size ($\approx$ 30) allows a further improvement in performance of $\approx 25\%$.

Tables A.1:A.7 show the memory statistics for the tiled and shackled variants of the seven versions of Cholesky factorization. In each case, data is shown for the number of load and store operations, the number of misses in the Level 2 cache and the miss ratio for the Level 2 cache. This data shows that , with the exception of a few cases with a large number of conflict misses, shackling uniformly reduces the miss ratios for the Level 2 cache by an order of magnitude or more compared to the

```
do i = 1, n
  norm = 0
  do j = i, n
    norm = norm + A(j,i) * A(j,i)
  norm2 = dsqrt (norm)
  asqr = A(i,i) * A(i,i)
  A(i,i) =
    dsqrt(norm-asqr+((A(i,i)-norm2)^2))
  do j = i+1, n
    A(j,i) = A(j,i) / A(i,i)
  do j = i+1, n
    x(j,i) = 0
    do k = i, n
      x(j,i) += A(k,i) * A(k,j)
  do j = i+1, n
    do k = i+1, n
      A(k,j) = A(k,j) - A(k,i) * x(j,i)
```

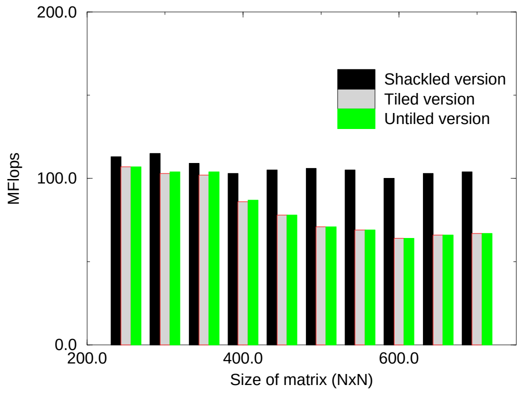


Figure 3.39: QR Factorization using Householder reflections

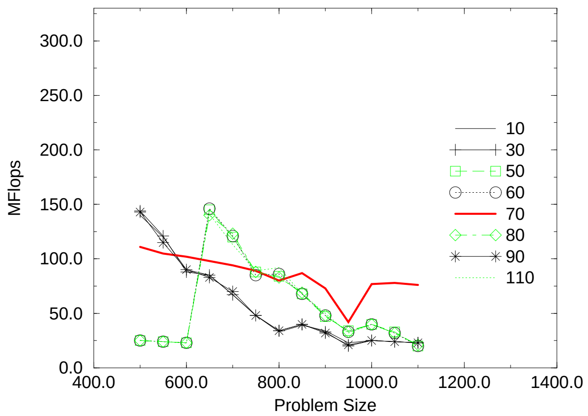


Figure 3.40: Performance of shackled ijk Cholesky for various block sizes

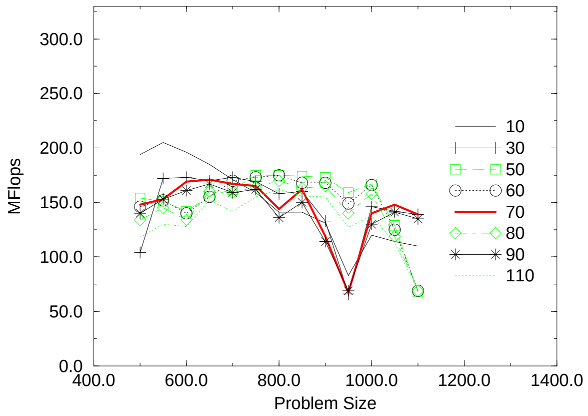


Figure 3.41: Performance of shackled ikj Cholesky for various block sizes

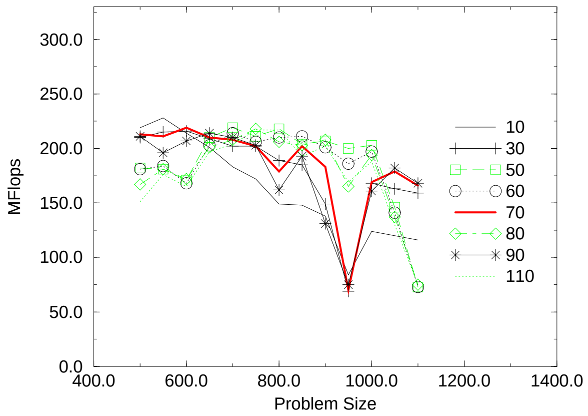


Figure 3.42: Performance of shackled jik Cholesky for various block sizes

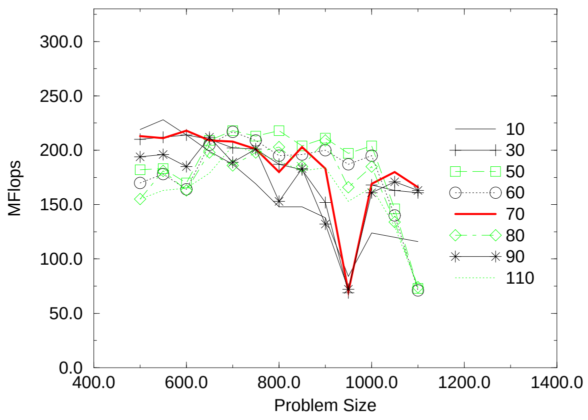


Figure 3.43: Performance of shackled jki Cholesky for various block sizes

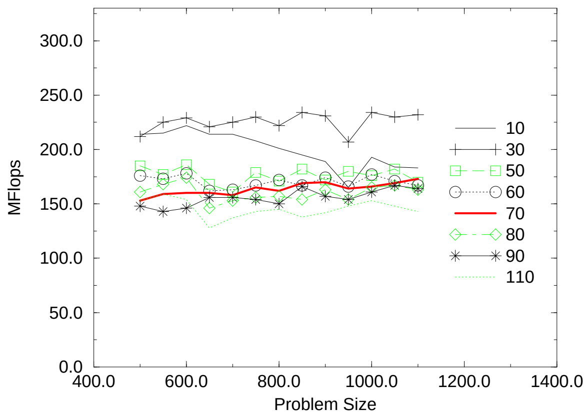


Figure 3.44: Performance of shackled kij (distributed) Cholesky for various block sizes

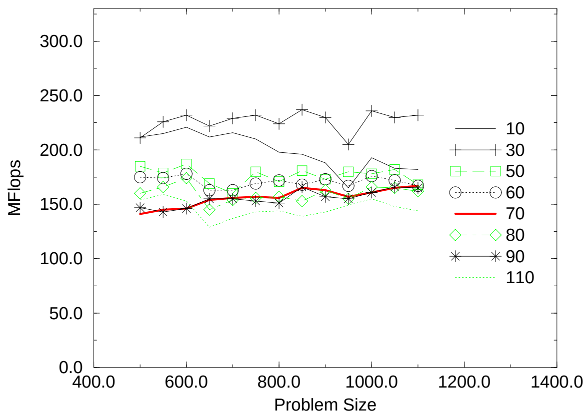


Figure 3.45: Performance of shackled kij (fused) Cholesky for various block sizes

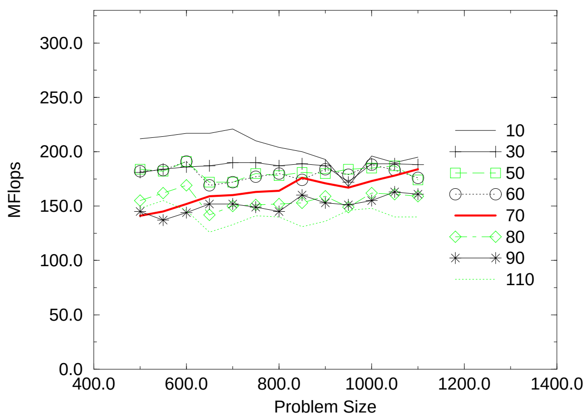


Figure 3.46: Performance of shackled kji Cholesky for various block sizes

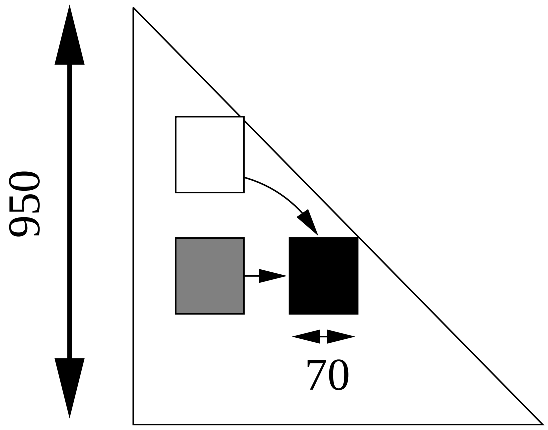


Figure 3.47: Explaining conflict misses in Cholesky Factorization

tiled code.

An interesting phenomenon in the performance results is the dip in performance of the shackled code for the `jik`, `jki`, `ijk` and the `ikj` versions when the size of the matrix is 950. This drop in performance is caused by a sudden increase in the number of L2 cache misses (as shown in Tables A.1:A.4). Detailed experiments with a cache simulator reveal that this sudden increase is caused by conflict misses. Note that the block size determination heuristic employed for shackling in the compiler is based purely on capacity considerations, and does not try to reduce conflict misses.

Figure 3.47 provides an explanation for the large number of conflict misses for this combination of problem size and block size. Most of the computation in Cholesky Factorization is in the update of off-diagonal blocks. On the Octane workstations, the L2 cache is 1MB in size and is organized as a 2-way set associative cache of 8192 lines, where each line is 128 bytes in size. Since the matrix under consideration has 950 rows (and columns), and is stored in column-major format, two elements in the same row, but adjacent columns have addresses that differ by 950*8 = 7600 bytes. In other words, successive elements in the first row of the black block in Figure 3.47 have addresses that differ from each other by 7600 bytes. Two addresses differing by 7600 bytes map onto cache lines separated by a distance of 7600/128 $\approx$ 60 lines. Since the black block in Figure 3.47 has 70 columns, the separation between the start address of the first element in the first column and the first element in the last column corresponds to (70-1)*60 $\approx$ 4200 lines. Since cache lines separated by a distance of 4096 map onto the same set, this implies that there is some self interference for this combination of problem and block size between the start and end of the block being updated.

However, self-interference by itself is not enough to explain the large number of conflict misses. This is because the two-way set associativity of the L2 cache takes care of this self-interference. However, the lightly-shaded block in Figure 3.47 is mapped into the L2 cache in a manner similar to the black block. Indeed, as the black block is successively updated from the left, all the blocks in the same block row all have the property that their start and end have self interference. Since each block starts immediately after the end of the preceding block, this interference in fact occurs for all blocks to the left of the black block. The overall effect is a massive increase in the total number of conflict (and cache) misses.

Figures 3.40:3.43 suggest a solution. Choosing a block size smaller than that suggested by capacity considerations alone alleviate the problem of increased conflict misses for three of the four versions of Cholesky Factorization. However, a systematic mechanism to deal with conflict misses still needs to be developed, and is beyond the scope of this dissertation.

3.5 Summary of the Data-centric Approach

Chapter 2 demonstrated the inadequacy of existing compiler approach in effectively enhancing the locality of general matrix factorization codes found in LAPACK. Existing compiler approaches are very effective for a simple class of programs, called perfectly nested loops. While this is adequate for simple dense matrix algorithms representing computations found in the BLAS library, computations in LAPACK are non-perfectly nested. Attempts to extend the perfectly nested loop based compiler technology to non-perfectly nested loops have not been very successful, as discussed in Chapter 2.

This chapter presents an alternative approach to the problem of locality enhancement, the *data-centric* approach. The driving force behind this approach is that iteration-space approaches represent an indirect attempt to improve locality, and a more direct solution to the problem of locality enhancement can be obtained by concentrating on the data domain instead. The key idea in the data-centric approach is to logically partition an underlying data structure, visiting each data block in this partition by turn, and executing work associated with each data block when it is visited. The *data shackle* mechanism is used to specify each of these steps. In the dense linear algebra domain, the objects of interest are multidimensional arrays, which can be partitioned into blocks using parallel equidistant sets of cutting planes. This partitioning also imposes a simple order on the resulting blocks, which can be used to enumerate them. Finally, work is associated with each block by picking a *data-centric* reference from each statement, so that when a block of data is visited, only those instances of the statement are executed for which the data touched by the data-centric reference lies within the current block.

The data shackle constitutes a simple approach to the problem of locality enhancement as compared to iteration-centric approaches. In particular, the data-centric approach is not concerned whether the program under consideration is perfectly nested or not, thus overcoming the a crucial limitation suffered by prior compiler approaches. An analogy can be drawn between applying the data-centric approach to locality enhancement and the use of Fourier transforms in signal processing. The Fourier transform allows for simpler solutions to problems in the signal processing domain by mapping a problem in the time domain to a problem in the frequency domain. Similarly, the data-centric approach solves the problem of locality by mapping it into the data domain, where it can be solved more efficiently, instead of being limited by the structure of the input program as is the case with the linear loop transformation framework for perfectly nested loops.

The simplicity of the data-centric approach has allowed it to be integrated into the production compiler at Silicon Graphics(SGI). This has required that simple heuristics be developed to determine the components of a data shackle, such as the orientation of cutting planes, block sizes and choice of data-centric references. Detailed performance evaluation has been presented in this chapter to demonstrate that for all variants of Cholesky Factorization, LU Factorization with Pivoting, and

QR Factorization with Householder Reflections, a relatively simple implementation of shackling significantly outperforms the state-of-the-art implementation of tiling in the SGI compiler.

Chapter 4

Data-shackling for Parallelization

4.1 Introduction

The data-centric approach for locality enhancement described in Chapter 3 is similar to the runtime and compile-time resolution approaches to compiling shared memory programs for distributed memory machines [40, 41]. This raises the following question: can the data shackle introduced as a mechanism to enhance locality automatically be also used by a parallelizing compiler? This chapter addresses this question for a special class of applications - semi-structured applications, including such key computations as multigrid methods and adaptive mesh refinement. The work described in this chapter was published in [14].

4.2 Semi-structured methods

Semi-structured methods such as adaptive mesh refinement and multigrid are used in applications which are computationally intensive. It is difficult to implement these methods efficiently even on a sequential machine; parallelism adds an order of magnitude overhead to the complexity.

The computation in semi-structured methods is characterized by irregularly organized regular computations on the underlying data. The underlying data is specified as *grid components* which are organized in an irregular fashion in a *grid hierarchy* which itself changes dynamically. The computation in the application consists of stencil operations for relaxation on grid components, and interpolation and projection operations for transferring data between grid components at different levels of the grid hierarchy. The computation is regular (that is, data access functions are affine functions of surrounding loop indices). Parallelism is obtained by distributing the grid components in the hierarchy among various processors. However, this also leads to communication between various grid components. The volume and patterns of communication depend critically on the data distribution. The irregular and evolving nature of the application also leads to load imbalance, requiring redistribution of data at run time.

Even though the underlying computation is dense, HPF-like compiler technology is inadequate for AMR applications. There are two reasons for this:

- Block-cyclic distributions are inadequate to obtain proper load balance for these applications. Obtaining good load balance requires grid components to be distributed in complex ways among the processors, as explained below.

- The communication between various grid components is determined by the connectivity of the grid hierarchy. Due to the irregular nature of this connectivity, communication patterns are irregular.

Because of the inadequacy of present compiler technology, several libraries have been developed to make the application programmer's job easier. The most important of these are Multi-block PARTI [2], P++/AMR++ [6], LPARX [5], DAGH [36]. All these libraries hide the nature of the data distribution from the application programmer as much as possible. They provide constructs such as *forall* loops to enable application writing at a high level, and provide library calls to take care of parallelization issues. Each of these libraries makes certain assumptions regarding how the underlying data is distributed, which affects application performance. *Multi-block PARTI* [2] uses block-cyclic distributions as in HPF, but allows data arrays to be mapped to a subspace of all the computing processors. This works well for Multigrid codes, but for adaptive mesh refinement, the limitations of HPF apply here as well. *AMR++* [6] is an AMR class library layered on top of a parallel array library P++. AMR++ treats P++ as a black box and uses the distributions that P++ provides. These distributions reduce load-imbalance by allowing arrays to be distributed by columns of variable size, but are still limited in expressiveness. *LPARX* [5] allows data arrays to be distributed in irregularly shaped and irregularly sized blocks onto processors. While in principle it is possible in LPARX for multiple blocks of data to be assigned to one processor, in practice there is only one data block per processor. For adaptive mesh refinement, LPARX helps in reducing the communication between grid components at the same level, but the communication between grid components at different levels increases. Finally, *DAGH* [36] uses a space-filling curve enumeration to distribute the blocks of an array onto processors. Space filling curves ensure spatial locality, which means that this distribution policy reduces communication between grid components at different levels of the grid hierarchy.

From an examination of all the above libraries, it is clear that there is no single universal distribution policy that is superior to all others. This makes it hard for the application programmer to experiment with different distribution policies. The libraries also need to be extended when they do not satisfy the users' needs. For example, DAGH comes with predefined stencils for relaxation schemes which cover many, but not all relaxation methods. Successful compiler support here would be of benefit to the application writers as well as library writers.

To this end, there have been some generalizations of the distributions allowed in HPF, the most general of which is the generalized block distribution in HPF-2 [21]. The generalized block distribution for an array is defined by a cartesian product of intervals, where each interval partitions one dimension of the array. This generalization captures the distributions in libraries such as Multi-block PARTI and AMR++, but it is inadequate to express distributions in libraries such as LPARX and DAGH.

This chapter demonstrates the use of the *data shackle* to automatically parallelize some of the computations in adaptive mesh refinement. The *fully general block distribution* is proposed to capture all the distributions used in the libraries mentioned above. The data-centric approach is extended to solve the standard challenges encountered by a parallelizing compiler for this class of distributions. Data structures from computational geometry are described that enable efficient manipulation of information about these distributions. Finally, modifications to these techniques that allow the handling of dynamic data distributions efficiently are also presented.

The rest of the chapter is organized in two parts. Section 4.3 describes the new data distribution. Section 4.4 discusses how parallel code can be generated if the parameters of the data distribution are known completely at compile time. Section 4.5 describes how these techniques are modified when data distributions change dynamically. Section 4.6 describes incremental algorithms for computing communication sets. Preliminary performance results are presented validating the extension of the data-centric approach for automatic parallelization.

4.3 Fully general block distributions

Intuitively, a fully general block distribution partitions the elements of an array into rectangular blocks whose sides are parallel to the data coordinate axes. Different blocks can have different sizes. More formally, a block is defined as follows.

A *block* in d-dimensions is a rectangular parallelopiped in d-dimensions, and can be specified by two vectors: $\vec{o} = [o_1, \ldots, o_d]$ and $\vec{e} = [e_1, \ldots, e_d]$, where $\vec{o}$ represents one of the vertices of the parallelopiped, and $\vec{e}$ represents the "extent" of the parallelopiped in each of the d-dimensions. By requiring that all the entries of $\vec{e}$

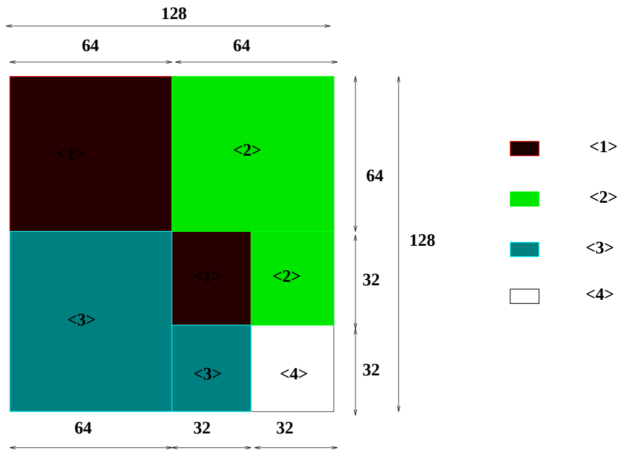


Figure 4.1: Example of a valid fully general block distribution

be non-negative, one obtains a canonical representation of a block. A *fully general block distribution* for a data item D is specified as a set of tuples of the form (B_i, P_i), where B_i is a block as defined above, and P_i is a processor assignment of this block. In addition, it must be the case that every element of D must occur in precisely one block, and any element of a block B_i must be a valid element of D (*i.e.*, it must be within array bounds of D). Note that the partitions of a data object into blocks using sets of cutting planes (as used by a data shackle to enhance locality) are a special case of a fully general block distribution.

Figure 4.1 shows an instance of a *fully general block distribution.* The general block distributions described in the literature [20, 21, 22, 50] are special cases of this distributions.

4.3.1 Distribution descriptor

A *distribution descriptor* is associated with each distributed data item. This descriptor is a data structure which provides information about the distribution of the associated data item in a structured form. In particular, the following information is provided to each processor.

- Number of blocks of the data item on a particular processor
- For each block, the global indices of the data object contained in the block, specified by the origin ($\vec{o}$) and extent ($\vec{e}$) vectors.

It is reasonable to assume that this information is replicated; *i.e.*, that every processor has access to the complete distribution information of the data item under consideration. This is because in a semi-structured application such as adaptive mesh refinement or multigrid, the uniprocessor performance is derived from working on large blocks, which implies that the distributed data structure is partitioned into a relatively small number of blocks.

A particular representation for the distribution descriptor is as an ordered set of sets each describing the local allocation of each processor. For example, the distribution descriptor for the distribution in Figure 4.1 consists of the four sets $\{([1,1], [64,64]), ([65,65], [32,32])\}$, $\{([1,65], [64,64]), ([65,97], [32,32])\}$, $\{([65,1], [64,64]), ([97,65], [32,32])\}$ and $\{([97,97], [32,32])\}$ for processors 1, 2, 3 and 4 respectively.

A fully general block distribution is clearly more complex to specify than a block or cyclic distribution in HPF. However, it appears that in most AMR applications,

```
Atp1 and A are N-by-N matrices
do i = 2, 127
  do j = 2, 127
    S1: Atp1(i,j) = c* (A(i,j+1)+A(i+1,j)+A(i-1,j)+A(i,j-1)-4*A(i,j))
```

Figure 4.2: Relaxation code in 2-d wave equation

the distributions of data arrays at the start of program execution are usually very simple, but become complex during program execution. Therefore, the fully general block distribution needs to be supported by the compiler and runtime system, but it may not be necessary to have directives in the source language to specify such distributions.

For the rest of the chapter, the relaxation code in Figure 4.2 is used as the running example, with the data distribution for `A` and `Atp1` as specified in Figure 4.1. This is a simplified version of the actual relaxation code that is used in the solution to the 2-D wave equation using the DAGH library [36, 18], and illustrates all the principles behind the extension of data shackling for automatic parallelization.

4.3.2 Issues for a Parallelizing Compiler

Given a program with data distribution specifications, a restructuring compiler performs the following tasks:

- Assignment of computations to processors.
- Generation of code to enumerate local iteration sets.
- Local storage allocation for distributed arrays.

- Determination of communication sets.
- Placement of communication in the program.

For dense programs, block and cyclic distributions (as in HPF [20]) are standard, and a simple rule like the *owner-computes* rule [40] is used to determine the iterations to be performed on each processor. In this case, closed form linear integer constraints can be used to express the local storage requirements, the local iteration sets, communication sets as well as the placement of communication. While the efficient placement of communication is a major concern for programs with dependences, it is straightforward in the case of *do-all* loops, since all communication for the loop nest can be performed before the loop nest begins execution. Communication optimizations such as combining communication from different loop nests to reduce the overall communication volume are not dealt with here. The rest of this chapter addresses the remaining four problems listed above for the case of fully general block distributions.

4.4 Parallelization in the static case

First the static case is examined (*i.e.*, when the parameters of the distribution are known at compile time). This is the case in codes such as multi-block Euler solvers [13]. A more general treatment of some of the material of this section can be found in [28].

```
Atp1 and A are N-by-N matrices
do i = 2, 127
  do j = 2, 127
  if i'm data-centric w.r.t. A(i,j+1)
  if ((97 ≤ i ≤ 128) && (97 ≤ j+1 ≤ 128))
    S1: Atp1(i,j)= c*(A(i,j+1)+A(i,j-1)+A(i-1,j)+A(i+1,j)-4.0*A(i,j))
```

Figure 4.3: Naive code for processor 4 with `a(i,j+1)` as data-centric reference.

4.4.1 Local iteration Set

For any loop nest, every processor must determine the set of iterations it will execute. This requires the determination of the *computation decomposition*. The data shackle provides a natural means to accomplish this. A single reference to the distributed array is chosen as the data-centric reference for all the statements in a loop nest, and only those iterations of the loop nest are performed for which the data touched by this reference are local to the processor. This is analogous to the use of the data shackle for locality, which requires that the data touched by the data-centric reference for each assignment statement be within the block of data currently visited. However, choosing a single reference as the data-centric reference for all the statements in a loop nest is more restrictive than in the case when the data shackle is used for locality enhancement. This is illustrated with an example.

`A(i,j+1)` is chosen as the data-centric reference for the loop nest in the running example. This causes processor 4 (which owns the block(97..128, 97..128) of `A`) to execute the code in Figure 4.3. Note that this code is still written in a *shared-memory* style, because the indices used to access the elements of all the arrays are

```
do i = 97, 127
  do j = 96, 127
    s1: Atp1(i,j)= c*(A(i,j+1)+A(i,j-1)+A(i-1,j)+A(i+1,j)-4.0*A(i,j))
```

Figure 4.4: Simplified code to be executed by processor 4

global indices. In addition, as in the naive code in the case of locality enhancement, this code is similar to that produced by *run-time resolution* [40] for distributed memory multiprocessors. As before, the affine conditionals introduced to force the data touched by the data-centric reference to be on the local processor are referred to as *localization constraints*.

Since all *localization constraints* are simple affine constraints on surrounding loop indices and constants representing block bounds, using technology presented in Section 3.3, they can be simplified and folded into loop bounds. Doing this produces code shown in Figure 4.4.

It is possible for a processor to contain multiple blocks of the data item used to partition computational work (for example, processor 1 in the running example). Since the distribution descriptor is available at compile time, one of two approaches can be used: (i) generate a different loop nest for each local block, or (ii) generate an outer loop that enumerates local blocks, with inner loops generating the iteration set for each block. The code generated for processor 1 using approach (i) is shown in Figure 4.5(i). Note that this approach cannot be used if the descriptor is not known to the compiler. Figure 4.5(ii) shows the code generated for processor 1 using the second approach.

(i) One possible program for a processor containing multiple blocks of A

```
do i = 1, 64
  do j = 1, 63
    s1: Atp1(i,j) =c*(A(i,j+1)+A(i,j-1)+A(i-1,j)+A(i+1,j)-4.0*A(i,j))
do i = 65,96
  do j = 64, 95
    s1: Atp1(i,j) =c*(A(i,j+1)+A(i,j-1)+A(i-1,j)+A(i+1,j)-4.0*A(i,j))
```

(ii) Alternative program for the same processor

```
Let D(A) be the distribution descriptor of A
D(A).first is first block of A, D(A).last is last
lo$dim and hi$dim are bounds for dimension $dim
DO bl = D(A).MYPROC.first, D(A).MYPROC.last
  DO i = max(D(A).lo1(bl), 2), min(D(A).hi2(bl), 128)
    DO j = max(D(A).lo2(bl), 2)-1, min(D(A).hi2(bl), 128)-1
      S1: Atp1(i,j)=c*(A(i,j+1)+A(i,j-1)+A(i-1,j)+A(i+1,j)-4.0*A(i,j))
```

Figure 4.5: Possible programs for a processor with multiple blocks of A

4.4.2 Local storage allocation

It is easy to see that while the reference that while the data-centric reference in a loop nest accesses only local data, other references in the loop nest can necessitate access to non-local data. In such a case, local storage needs to be allocated to store all the data items that a processor requires. This section discusses the analysis required for local storage allocation.

Let A be distributed in a fully general block distributed manner onto a set of processors. Let A_{pj} denote block j of A on processor p. Let R be the data-centric reference to A for the loop nest under consideration. Let $\mathfrak{F}(r)$ be the access matrix [49] for any reference r, i.e. iteration $\vec{i}$ touches data item $\mathfrak{F}(r) * \vec{i}$ through reference r. The following definitions are useful:

Definition 16 *The* per-block owned data $\mathfrak{OB}(p, A_{pj})$ *of a block is the elements of the data item it contains.*

Definition 17 *The* per-block local iteration space $\mathfrak{LB}(p, j)$ *is defined as*

$$\mathfrak{LB}(p, A_{pj}) = \{\vec{i} | F(R) * \vec{i} \in \mathfrak{OB}(p, A_{pj})\} \tag{4.1}$$

Definition 18 *The* per-block per-reference view set *for block j for reference r on processor p, written as $\mathfrak{vB}(r, p, j)$, is defined as*

$$\mathfrak{vB}(r, p, A_{pj}) = F(r) * \mathfrak{LB}(p, A_{pj}) \tag{4.2}$$

Definition 19 *The* per-block view set *for block j on processor p for* `A`*, written as $\mathfrak{VB}(p, A_{pj})$, is defined as*

$$\bigcup_r \mathfrak{vB}(r, p, A_{pj})$$

where r ranges over all the references to A in the loop nest.

The *per-block view set* represents the set of elements that are required for the execution of a block for the given loop nest. In general, this set is arbitrarily shaped. The smallest enclosing isothetic rectangle enclosing this set is determined and storage allocated for this rectangle on the local processor.

The computation of view sets proceeds as follows: Consider block A_{12}, represented by the tuple $\{([65, 65], [32, 32])\}$. Let `A(I,J+1)` be the data-centric reference. $\mathfrak{LB}(1, A_{12})$ is the set of iterations for which I is in the range $65 \ldots 96$, and J is in the range $64 \ldots 95$. The *per-block per-reference view set* for A_{12} for the references `A(I,J)`, `A(I+1,J)`, `A(I-1,J)`, `A(I,J+1)` and `A(I,J-1)` are $\{([65, 64],\ [32, 32])\}$,

```
DO i = 65, 96
  DO j = 64, 95
    S1: Atp1(i,j) = c* (lA12(i-63,j+1-62)+lA12(i-63,j-1-62)+
          lA12(i-1-63,j-62)+lA12(i+1-63,j-62)-4.0*lA12(i-63,j-62))
```

Figure 4.6: Code executed by processor 1 after translation to local index sets

$\{([66,64],[32,32])\}$, $\{([64,64],[32,32])\}$, $\{([65,65],[32,32])\}$ and $\{([65,63],[32,32])\}$ respectively. The smallest rectangle that encloses the union of these five view sets is the set $\{([64,63],[34,34])\}$. This is precisely what the *ghost region* support in the libraries such as LPARX and DAGH do and is a generalization of overlap analysis introduced by Gerndt [23]. Note that this storage can be determined automatically given the data distribution. In particular, no information about the type of stencils used, etc. need be conveyed to the compiler, since the data usage information can be extracted from the input code. Note also that the storage allocated for each block on a processor is enough to contain the *view set* of the block, which is a *superset* of the elements *owned* by the block. Finally, note that if an element of an array occurs in the view sets of multiple blocks, space is allocated for it multiple times.

Global and Local index translation

In the node program on each processor of a distributed memory machine, all array accesses must be to local indices. The storage for the view set of each block introduces a specific index translation from the global indices to local indices. Each local index of a given block is at a fixed offset from the corresponding global index, which is determined by the global index of the *first location* of the view set.

For example, the view set of A_{12} requires a local array of size 34-by-34 elements. Element (1,1) of this local array (call it lA12) corresponds to element A(64,63) of the global array. Consequently, a local index of A_{12} is related to the corresponding global index by the offset vector $(63, 62)$. Given this information, it is easy to see that the enumeration of the *per-block iteration space* for A_{12} corresponds to Figure 4.6.

4.4.3 Communication sets

The final step in the parallelization process is one in which every processor determines what data to send and what data to receive before the start of execution of a loop. The following definitions are useful:

Definition 20 *The* owns set *for data item* `A` *on processor p,* $\mathfrak{O}(p, A)$*, is defined as the union of* per-block owned data *sets of all blocks of A on p. In other words,*

$$\mathfrak{O}(p, A) = \bigcup_j \mathfrak{O}\mathfrak{B}(p, A_{pj})$$

where j ranges over all blocks of A on p.

Definition 21 *The* view set *for data item* `A` *on processor p, written as* $\mathfrak{V}(p, A)$*, is defined to be the union of the* per-block view sets *over all the blocks of* `A` *on p. In other words,*

$$\mathfrak{V}(p, A) = \bigcup_j \mathfrak{V}\mathfrak{B}(p, A_{pj}),$$

where j ranges over all blocks of A on p.

Two processors p and q need to communicate elements of A if $\mathfrak{V}(p, A) \cap \mathfrak{O}(q, A) \neq \phi$. Let $\mathfrak{S}(p, q, A)$ denote the elements of `A` that processor p needs to send to processor q. Let $\mathfrak{R}(p, q, A)$ denote the elements of `A` that processor p needs to receive from processor q. Then,

$$\mathfrak{S}(p, q, A) = \mathfrak{R}(q, p, A) = \mathfrak{O}(p, A) \cap \mathfrak{V}(q, A) \tag{4.3}$$

Once the communication sets are available, a generic communication routine enumerates over all the elements of these sets and performs the appropriate send (and recv) operations. In Section 4.6, a fast algorithm for computing the communication sets efficiently is presented.

4.5 Modification for dynamic and adaptive distributions

In general, the grid hierarchy may be known only when the program begins execution (dynamic case), and it may even be modified during the course of program execution (adaptive case). How does this affect the techniques described in Section 4.4?

To enumerate over the local iteration space, the compiler must generate code similar to Figure 4.5(ii), rather than Figure 4.5(i). Local storage allocation and computation of communication sets is dependent on knowing the data distribution. While the compiler cannot compute the appropriate sets at compile time, it is easy to write code that computes these sets at runtime and performs storage allocation and communication as needed. This code is part of the runtime library. Before each loop nest begins execution, the library code determines communication sets, and a generic communication routine enumerates over all elements of the communication sets, performing the appropriate send (and recv) operations.

There are two considerations if the data distributions of arrays can change at run time:

1. Since data descriptors are replicated, they must be updated consistently across all processors.

2. A more important concern is the efficiency of computing information such as communication sets. In the static or dynamic case, these sets are computed once, so efficiency of this computation is not a major issue. In the adaptive case, this computation is performed repeatedly, so it is important that it be performed efficiently. This problem is addressed next.

4.6 Incremental Communication Set Generation and Maintenance

Data redistribution in the adaptive case is usually incremental, so an incremental computation of communication sets is preferable to complete recomputation. Techniques from computational geometry can be used to accomplish this.

Consider the case when the *per-block per-reference view sets* are all rectangular parallelopipeds. This will be the case for any reference for which the access matrix can be written as the product of a diagonal matrix and a permutation matrix. The exact set of indices that need to be communicated between every non-local block which *owns* some data in the *per-block per-reference view set* of some local block is the intersection of the rectangles representing these two sets. The general problem to determine the communication for a single block is the following: Given a set of input blocks on all processors representing data ownership, determine the intersection of the *per-reference per-block view set* rectangle with the input set. This is a variant of the well known multidimensional range-search problem in computational geometry [33]. The rectangles representing data ownership correspond to *input rectangles* and the rectangle representing the *per-reference per-block view set* is called the *query rectangle.*

There is a naive solution to the above problem - enumerate over all the rectangles in the input set and test each rectangle for intersection with the query rectangle. Two rectangles in d- dimensions intersect if and only if their projections on all d axes intersect. Evaluating this takes $O(d)$ time. Thus, a naive solution to the query formulated above takes $O(Nd)$ time, where N is the number of input rectangles (the number of blocks of data distributed).

Pre-processing the input set of rectangles representing ownership information, as described below, allows this query to be answered in time $(log^d(N) + k)$ [34], where k is the number of rectangles that intersect. In other words, given a certain number of input rectangles, the query time grows as the size of the actual number of positive results for the given query, rather than the size of the input set. Since these queries have to be solved quite often, the savings can be substantial in large problems.

Finally it should be noted that every time a query intersection with the input set of rectangles is performed, the results can be *cached* . For identical queries in the future, the result of the earlier evaluation can simply be reused. This can help save on computations of view sets for the same reference in different loop nests, for example.

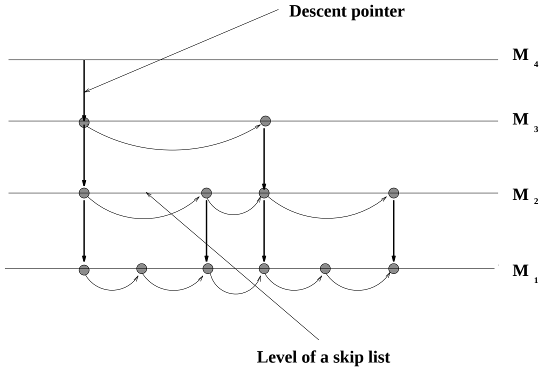


Figure 4.7: An example skip list

4.6.1 Orthogonal range-search problem

The *orthogonal range-search problem* is defined as follows: In d- dimensions, an orthogonal object is defined to be simply the cartesian product of d intervals. Given an input set $\mathfrak{I}$ of input orthogonal objects and a query orthogonal object q, the solution to the *orthogonal range search problem* is required to report all objects in $\mathfrak{I}$ that intersect with q. The solution to this problem is discussed here for d equal to 1 - the general case is described in [34].

In 1-dimension, an orthogonal object is a line segment, which is completely specified by its endpoints. The problem then is, given a set of input line segments, to report all line segments from this set that intersect a query line segment. This is done as follows.

```
M is an input set of coordinates on the real line
M_1 = M, i = 1
while (M_i) is non-empty do
  create M_{i+1} = φ
  for (all elements e ∈ M_i do)
    randomcoin() is a fair coin
    toss = randomcoin()
    if (toss == head)
      M_{i+1} = M_{i+1} ∪ e
  i++
nlevels = i
for j = nlevels, 1, -1 do
  for (all elements e ∈ M_j do)
     create descent pointer to element at same location
```

Figure 4.8: Procedure for constructing a skip list from a set of input coordinates

Randomized skip list

The first step is to pre-process the input line segments into a data structure that can be queried efficiently. The particular data structure chosen here is called the randomized skip list. Let M be any given set of m points on the real line R. Given a fair coin, a randomized search structure is associated with M. Starting with M, a sequence of sets $M = M_1 \supseteq M_2 \supseteq M_2 \supseteq M_{r-1} \supseteq M_r = \phi$, where set M_{j+1} is obtained from set M_j by tossing the fair coin for each point in M_j and including only those points in M_{j+1} for which the toss results in a head. This sequence of sets is known as a *gradation*. The expected number of levels in the gradation is $O(log\ m)$. Each level of the partition is stored as a linked list of the points in that level in ascending order. In addition, a *descent pointer* is maintained from each point in a level to the point in the immediately lower level with the same coordinate.

```
L is the skip list, and q is a query coordinate
identify the interval at every level of L containing q
level = toplevel; child = descent pointer from top level

q is located trivially in the top most level
for i = nlevels-1, 1, -1 do
  parent = interval containing q in level i+1
  left = left end point of parent;
  ld = descent pointer to level i from left
  j = ld
  while (coordinate of point j in level i < q)
    j = next point at level i
  store [j..j+1] as the interval containing q at level i
```

Figure 4.9: Procedure to determine the intervals of skip list containing a query point

The topmost level of the gradation (which is empty) maintains a single pointer to the first point in the next lowest level. This storage mechanism for the gradation is known as a *skip list* and is shown in Figure 4.7. It may also be noted that the points in every level of the skip list break up the real line into a set of intervals. $\mathfrak{H}(M_i)$ denotes the set of intervals generated by level i. Figure 4.8 shows the procedure for constructing a skip list from an input set of points. A skiplist can be used for searching using the code in Figure 4.9. The expected cost of the search procedure is equal to the number of children of intervals in the various levels of the skip list that contain the query point. It is straightforward to show this to be equal to $O(log\ m)$.

Augmented skip list

The skip list needs to be augmented to answer queries about segment intersections, certain augmentation must be performed. Given s input segments in set S, the set M consists of $2 * s$ elements and is formed by taking all the endpoints of all the segments of S. A skip list is formed using the procedure described in section 4.6.1. An interval of a skip list is defined to be an interval of the real line that belongs to $\mathfrak{H}(M, i)$ for some i. The parent of an interval I at level i is defined to be the interval J at level $i+1$ that contains I. An input segment S_i of S covers an interval I of M if S_i contains I. S_i is defined to cover I *canonically* if S_i covers I and the parent of I covers S_i. The following lemma holds when S_i covers I canonically.

Lemma 1 *An interval S_i covers an interval I of a skip list* canonically *iff the parent of I contains an endpoint of S_i and S_i covers I.*

Thus, the intervals of a skip list covered canonically by a given input segment S_i can be determined by testing the children of the intervals of the skip list containing an end point of S_i

An *augmented skip list* is formed by storing with each interval I of a skip list, the list of input segments that cover I canonically.

The following theorem is stated without proof. A detailed proof can be found in [34].

Theorem 4 *A skiplist built on an a set of s intervals of set S can be augmented in $O(slog\ s)$ time and the augmentation requires $O(slog\ s)$ space.*

```
b is a block of data of array A
D is A's distribution descriptor, r is a reference
Let E be the set of segments in D
M is the skip list on endpoints of segments in E
R =  E  ∩ 𝔳𝔅(r, MYPROC, b)
for (all elements r of R  do)
  recv-section = r ∩ 𝔳𝔅(r,  MYPROC,  b)
  mpi-recv(owner(r), recv-section)
```

Figure 4.10: Code a processor executes to determine recv calls it must insert for a given block

Answering intersection queries and generating send/recv calls

How are queries regarding the intersection of a query segment with the set of input segments answered? It is easy to see that two segments intersect if one of them contains an endpoint of the other. When presented with a query segment, first all the input segments that canonically cover all intervals of the skiplist containing an end point of the query segment are reported. In addition, all the input intervals completely contained inside the query segment are also reported. This is done by locating the left endpoint of the query segment in the lowest level of the skip list, and traversing the level starting from there and reporting all segments for which both endpoints are within the query range. A more detailed discussion of this procedure and a generalization to d-dimensions is presented in [34]

How does this relate to doing sends and receives? The relationship is shown in one dimension - the case is similar for higher dimensions. Figure 4.10 shows how the results of the range search problem can be used to generate recv calls for a single block of data. The procedure for generating the send calls is analogous.

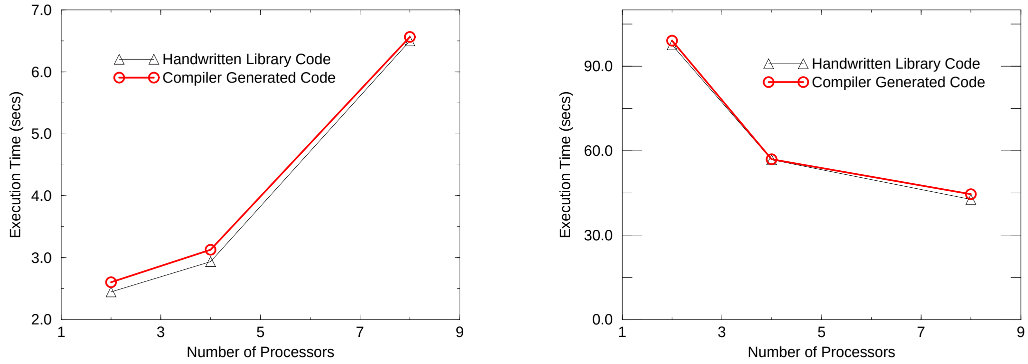
4.6.2 Dynamization

The skip list allows for efficient dynamization. Segments can be deleted and inserted into the skip list very efficiently. This allows for incremental maintenance of distribution information in the skip list when data distribution changes at run time. The following theorem (stated without proof) is crucial to the efficient dynamic behavior of the skip list. As usual, the theorem is proved in [34].

Theorem 5 *A single point can be inserted into or deleted from a skip list in time $O(log\ m)$, where m is the number of points in the skip list.*

4.7 Performance

Finally, some preliminary performance results are presented in Figure 4.11. A 2-D multigrid solver was implemented for this purpose using the DAGH library with published code [12]. This is the line labeled "execution time for multigrid solver" in Figure 4.11. Next, a version of this solver, augmented with the actual intersection queries the compiler would insert to determine the communication sets at run time, was written by hand. This is the line labeled "execution time for solver with queries" in Figure 4.11. Since the compiler does not have any knowledge of specific data distribution policies (which the library does) and the distribution is unknown at compile time, the compiler-generated code must execute these queries. As can be seen, the resulting overhead is minimal both when the problem sizes involved are



(a) 17-by-17 mesh (b) 1025-by-1025 mesh

Figure 4.11: Performance of a Multigrid Solver: (a) small problem, (b) large problem

large or small. Two different problem sizes were used - a small mesh of 17-by-17 points and a large mesh of 1025-by-1025 points. Figure 4.11(a) actually exhibits a slowdown because of the small problem size, but this case also demonstrates that the absolute overhead introduced by the dynamic skiplist approach is small. Figure 4.11(b) is more realistic. Overall, the overhead introduced is within 7% for the small problem and less than 1% for the large problem.

The performance benefit of using an asymptotically optimal algorithm rather than the naive algorithm are quantified next. When the number of blocks of data is less than 60, the naive search algorithm is faster. For larger numbers of blocks, the asymptotic algorithm is faster, but there is the penalty of preprocessing. When there were 400 blocks for example, an average query using the naive algorithm takes approximately $13 \mu s$, while the optimal algorithm takes approximately $7 \mu s$. The preprocessing time in this case takes 2.34 milliseconds, which means that at least 390 queries must be made for the optimal algorithm to be useful. To obtain a perspective, in the multigrid implementation, a total of 1710 queries were made for the small problem size and 4290 queries were made for the large problem size. This suggests that while the performance advantage of the incremental algorithm is small, it is easy to code and worth implementing.

4.8 Summary

The data-centric approach for locality enhancement is similar to the distributed memory parallelization approach of runtime resolution introduced by Anne Rogers [40]. This immediately raises the question whether the data-centric approach for locality enhancement can also be used for parallelization purposes. This chapter extends the notion of data shackle and develops technology to parallelize semi-structured applications including adaptive mesh refinement and multigrid computations. It is shown that using this approach, code competitive with hand-written libraries can be derived for this class of applications.

Chapter 5

Conclusions and Future Work

This chapter summarizes the contributions of this dissertation and discusses open issues and potential avenues of future research.

5.1 Summary of Dissertation

As processor and memory speeds have steadily diverged on modern high-performance machines, exploiting locality in memory hierarchies has become crucial to keeping the processor fully utilized. Unfortunately, most algorithms in the dense linear algebra domain do not exploit locality in their natural form and perform poorly. Hand-written libraries alleviate this problem somewhat by providing portable implementations of common computations. In the context of dense linear algebra, libraries have followed a two-stage layered approach - (i) the BLAS library provided by individual vendors encapsulating core computations such as matrix multiplication and matrix vector product, and (ii) the LAPACK library containing hand-restructured algorithms layered on top of the BLAS library. However, the libraries involve significant development and maintenance effort and are specific to the dense linear algebra domain. This raises the question of developing compiler techniques using which it would be possible to achieve the performance of handwritten libraries using program automatic program restructuring techniques.

The state-of-the-art in compiler technology for locality enhancement in this domain is based on the linear loop transformation framework for perfectly nested loops. Perfectly nested loops are simple programs in which all the assignment statements are completely enclosed within all the loops. Two key loop transformations, *interchange* and *tiling*, allow a perfectly nested loop with poor locality to be transformed into a program with improved locality. The core computations in the BLAS library are perfectly nested, and for these programs, compiler technology is very effective - for example in the case of matrix multiplication, a combination of loop interchange and tiling can achieve 80% of the performance of the handwritten code. However, the more complex algorithms in LAPACK are all non-perfectly nested, and there is no systematic framework to convert a non-perfectly nested loop to a perfectly nested loop. Various ad-hoc strategies have been proposed, but are unsuitable for integration in a production setting. As a result, existing production compiler technology is not very effective for computations in LAPACK, and handwritten library code is *3-5 times* faster than what the compiler can achieve, as demonstrated in Chapter 2. This is an undesirable situation.

Chapter 3 presents an alternative approach to the problem of locality enhancement, the *data-centric* approach. The driving force behind this approach is that iteration-space approaches represent an indirect attempt to improve locality, and a more direct solution to the problem of locality enhancement can be obtained by concentrating on the data domain instead. The key idea in the data-centric approach is to logically partition an underlying data structure, visiting each data block in this partition by turn, and executing work associated with each data block when it is visited. The *data shackle* mechanism is used to specify each of these steps. In the dense linear algebra domain, the objects of interest are multidimensional arrays, which can be partitioned into blocks using parallel equidistant sets of cutting planes. This partitioning also imposes a simple order on the resulting blocks, which can be used to enumerate them. Finally, work is associated with each block by picking a *data-centric* reference from each statement, so that when a block of data is visited, only those instances of the statement are executed for which the data touched by the data-centric reference lies within the current block.

The data shackle constitutes a simple approach to the problem of locality enhancement as compared to iteration-centric approaches. In particular, the data-centric approach is not concerned whether the program under consideration is perfectly nested or not, thus overcoming a crucial limitation suffered by prior approaches. The simplicity of the data-centric approach has allowed it to be integrated into the production compiler at Silicon Graphics(SGI). This has required the development of simple heuristics to determine the components of a data shackle. Detailed performance evaluation has demonstrated that for all variants of Cholesky Factorization, LU Factorization with Pivoting, and QR Factorization with Householder Reflections, a relatively simple implementation of shackling significantly outper-

(i) One-dimensional Relaxation

```
do i = 1, n
  do j = 1, m
    A(j) = (A(j-1) + A(j) + A(j+1))/3.0
```

(ii) Pictorial Representation of Relaxation

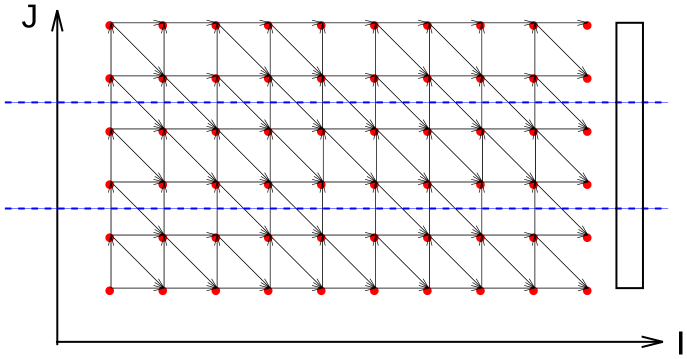


Figure 5.1: Shackling Relaxation code requires multiple passes over underlying array

forms the state-of-the-art implementation of tiling in the SGI compiler.

Finally, Chapter 4 extends the notion of data shackle to perform automatic parallelization. Technology is developed to parallelize semi-structured applications including adaptive mesh refinement and multigrid computations. It is shown that using this approach, code competitive with hand-written parallel libraries can be derived for this class of applications.

5.2 Future Directions of Research

As this dissertation has demonstrated, data shackling is a very powerful and novel technique for locality enhancement as well as parallelization. There are many interesting research questions that are open at the conclusion of this dissertation, which are well worth pursuing. This section attempts to provide a brief summary of these issues:

1. The first issue relates to addressing the performance gap that still remains between the code generated by shackling for LU Factorization with Partial Pivoting and QR Factorization when compared to the corresponding handwritten code. In each case, shackling is limited by the fact that the handwritten library code exploits domain-specific information that is not available to the compiler. For example, in the case of LU Factorization with Partial Pivoting, the domain specific information used in LAPACK is the knowledge that row-permutation operations commute with the update step. In the case of QR Factorization, the algorithm in LAPACK utilizes the fact that matrix product is associative to recast a series of Householder Reflection operations as a single Block-Householder Reflection with higher performance. A research question is whether a simple mechanism can be developed to impart this domain-specific information to the compiler, enabling it to generate code closer in performance to handwritten library code. One possible means to accomplish this would be to develop the Factorization codes in a language such as MATLAB, which has arrays and matrices as basic data types, potentially allowing this kind of

analysis.

2. The second interesting issue arises from the ease with which locality and parallelism can be addressed using the data-centric approach. This raises the possibility that the data-centric approach may present the appropriate framework for high-performance compilers on clusters of symmetric multiprocessors. These machines have multiple levels of parallelism across SMPs in a cluster as well as within an SMP; at the same time, exploiting locality within an SMP is crucial to achieving high performance. This dual demand on high performance compilers for this class of machines suggests that a single framework reasoning about locality and parallelism is more likely to be effective. The data-centric approach certainly holds this promise.

3. In all the LAPACK Factorization routines, and for all the examples encountered so far in the dense linear algebra domain, a single data shackle always makes a *single* pass over the underlying array. However, in general, multiple traversals of the underlying data structure may be necessary to be able to produce a legal shackle. A classic example of this scenario is the 1-dimensional relaxation code shown in Figure 5.1. The array `A` is one-dimensional, and can only be partitioned into one-dimensional slices by sets of cutting planes. However, no legal shackle can be obtained by choosing any of the three references, `A(j)`, `A(j-1)` or `A(j+1)`, if all the resulting work associated with every slice of `A` is required to be performed in one shot. This is because of the nature of dependences in the program, represented by directed arrows in Figure 5.1. A legal shackling can be obtained in this case only by performing multiple

traversals over the blocks of `A`, where only some of the work assigned to each slice of `A` is performed in each traversal. This generalization of data shackling to permit multiple traversals over a data structure adds more power to the framework, and it is an exciting question to determine the resulting benefit from the added expressiveness.

4. Another interesting problem domain where the data-centric approach may prove to have an impact is in FFT computations. The Kronecker product algebra for describing FFT computations [29, 26, 37], has led to many interesting results, where the basic FFT computation can be rewritten using a series of rewrite rules to exploit either parallelism or locality. However, relatively little is known about how to automatically derive the sequence of rewrite rules in a given architecture setting. It is conceivable that the data-centric approach provides a simple rationale to drive the application of these rewrite rules. This would have important implications in the embedded systems world, where FFT-like computations are very common. Typical examples are image transformation operations in digital cameras, and color space conversion operations in printers.

Appendix A

Memory Behavior of Shackling and Tiling for various versions of Cholesky Factorization

Table A.1: Memory statistics for tiled and shackled Cholesky (ijk version)

Tiling			Shackling			Miss ratios	
Loads	Stores	L2 misses	Loads	Stores	L2 misses	T	S
43787390	1157264	88676	35154945	16368218	41956	0.20	0.08
58052980	1379742	985962	45766495	22263420	47432	1.66	0.07
75129540	1621221	1152455	59382547	29433245	61433	1.50	0.07
95252833	1885653	1312406	74515299	38005836	75185	1.35	0.07
118658574	2169488	2010400	92761885	48105663	98582	1.66	0.07
145671415	2474240	2875584	112260877	59853845	116788	1.94	0.07
176594620	2801423	4110691	128075421	73380283	291681	2.29	0.14
211544943	3145992	5858019	164549300	87786914	204485	2.73	0.08
250846730	3519642	6201361	188717814	106257316	1181348	2.44	0.40
294717990	3906859	19627258	214881325	125860166	11074730	6.57	3.25
343463224	4317372	9496315	245516471	147740036	365460	2.73	0.09
397335645	4782275	10831998	301124326	172561295	500292	2.69	0.11
456557546	5200301	12736307	334807854	199418767	951103	2.76	0.18

Table A.2: Memory statistics for tiled and shackled Cholesky (ikj version)

Tiling			Shackling			Miss ratios	
Loads	Stores	L2 misses	Loads	Stores	L2 misses	T	S
49770012	4510541	158859	32879503	3075601	45476	0.29	0.13
65790766	5845836	1161629	41969300	3786873	58076	1.62	0.13
84861688	7300171	1436592	49553554	4472492	67270	1.56	0.12
107323126	8961091	1678301	62036087	5207400	78463	1.44	0.12
133370542	10925933	2453086	76374853	6416969	111515	1.70	0.13
163882700	13042674	3458636	93014789	7242283	145113	1.95	0.14
198545024	15050621	5185883	111812428	8500653	378419	2.43	0.31
231543826	16633527	6711787	132564683	10124177	221805	2.70	0.16
279418152	21058161	7829929	156474576	11218245	1726496	2.61	1.03
328661055	24294599	11409950	182862513	12860618	8811329	3.23	4.50
384008280	27259523	10723503	211221349	14894292	992941	2.61	0.44
442071660	31943855	12842843	243775006	17138766	737030	2.71	0.28
508886182	36322060	15203906	278538383	18629898	1386796	2.79	0.47

Table A.3: Memory statistics for tiled and shackled Cholesky (jik version)

Tiling			Shackling			Miss ratios	
Loads	Stores	L2 misses	Loads	Stores	L2 misses	T	S
25212338	1637316	82931	25369017	1707475	42904	0.31	0.16
33267819	2025553	178139	33396163	1859263	54424	0.50	0.15
42887721	2468639	174856	41469357	2649859	65462	0.39	0.15
54068100	2899185	501006	54415272	2720552	79388	0.88	0.14
67167349	3433457	626378	64962803	3679953	110229	0.89	0.16
82481967	4156228	1129511	80313202	4353546	131548	1.30	0.16
99502805	4712989	3620165	100098412	4370228	422910	3.47	0.40
119442646	5554174	3965079	115972862	6481326	241829	3.17	0.20
141493190	6389030	5592681	138405175	6616426	1702589	3.78	1.17
165837168	7550006	8466547	165940077	6427502	8935276	4.88	5.18
193290662	8286414	8828047	188620888	8666858	624870	4.38	0.32
223366654	9361292	10850315	223159800	8214782	672408	4.66	0.29
256572425	10522609	12885935	256250670	9211151	1023618	4.82	0.39

Table A.4: Memory statistics for tiled and shackled Cholesky (jki version)

Tiling			Shackling			Miss ratios	
Loads	Stores	L2 misses	Loads	Stores	L2 misses	T	S
25211170	1637315	65772	25369530	1707475	43389	0.24	0.16
33264741	2025553	207607	34518946	2513009	65355	0.59	0.18
42888684	2468643	133218	42540258	3195583	72340	0.29	0.16
54071701	2899185	562037	54174510	3167380	85995	0.99	0.15
67165966	3433461	508742	64963024	3679960	114538	0.72	0.17
82481819	4156231	1133762	82492076	5329544	153052	1.31	0.17
99502481	4712989	3621663	101985718	5108232	520821	3.48	0.49
119001818	5452490	3924965	115954779	6481325	206290	3.15	0.17
141497692	6389031	5589538	142444078	8410709	2250285	3.78	1.49
165844112	7550005	8493610	168832776	7909973	9645366	4.90	5.46
193289926	8286413	8824344	188088730	9796110	636680	4.38	0.32
223365619	9361293	10876625	223158976	8214781	585044	4.67	0.25
256569505	10522609	12849202	261304348	11528649	1295155	4.81	0.47

Table A.5: Memory statistics for tiled and shackled Cholesky (kij version, distributed)

Tiling			Shackling			Miss ratios	
Loads	Stores	L2 misses	Loads	Stores	L2 misses	T	S
17916874	21956468	132077	25891535	12922223	40279	0.33	0.10
25256704	29086469	961642	32515246	16952072	53844	1.77	0.11
31740152	37614488	1211926	42941882	21089289	59820	1.75	0.09
40900008	47673828	1318658	56047332	25990739	83097	1.49	0.10
48541644	59390283	2132898	70061864	31768427	95473	1.98	0.09
61225072	72885771	2967967	83026161	37929086	131467	2.21	0.11
72984732	88281755	4357161	101622328	45273297	178598	2.70	0.12
88596353	105479781	5878625	121039524	51952695	203294	3.03	0.12
104724889	125305276	6346053	140046915	62090525	261051	2.76	0.13
123301587	147173494	8550440	166643458	71628311	442877	3.16	0.19
136081208	171461285	9738997	198914233	82491658	351336	3.17	0.12
163798585	198285162	10973535	228106860	94403823	363948	3.03	0.11
182551519	227755949	12936778	259330902	106673948	452628	3.15	0.12

Table A.6: Memory statistics for tiled and shackled Cholesky (kij version, fused)

Tiling			Shackling			Miss ratios	
Loads	Stores	L2 misses	Loads	Stores	L2 misses	T	S
27635659	22619470	40856	25979180	12943297	42036	0.08	0.11
35286035	30898713	55460	32467363	16974302	53162	0.08	0.11
50158707	38927115	60276	42717982	21059559	64755	0.07	0.10
58032094	50780817	87914	56189343	26016005	78082	0.08	0.09
74053891	61158423	96165	69872726	31767322	96182	0.07	0.09
88458631	78100023	130743	82812811	37886847	125838	0.08	0.10
112098491	93885488	166567	101700433	45335281	188390	0.08	0.13
122563924	108106492	214468	120810226	51951866	190321	0.09	0.11
153029739	133146265	241567	139892580	62086948	264797	0.08	0.13
177252531	156095657	510306	166891217	71632823	442704	0.15	0.19
214218395	183260458	292373	198142900	82479964	349589	0.07	0.12
238181610	209831482	359001	228134397	94455236	365303	0.08	0.11
278578965	242997801	408340	259405891	106675769	432971	0.08	0.12

Table A.7: Memory statistics for tiled and shackled Cholesky (kji version)

Tiling			Shackling			Miss ratios	
Loads	Stores	L2 misses	Loads	Stores	L2 misses	T	S
43131031	21745202	173646	22983712	13362073	42685	0.27	0.12
57991667	29123389	1198987	30038179	17005779	53935	1.38	0.11
76731076	37655376	1504767	37096753	20918031	64631	1.32	0.11
95417758	47673270	1706315	48586601	26217268	77712	1.19	0.10
117739717	59338154	2549194	61052557	31537239	100988	1.44	0.11
146199140	73064675	3514990	72367472	37641793	122242	1.60	0.11
180452901	88387696	4859308	88546175	45273947	185307	1.81	0.14
212406207	105741780	6546307	104284228	52607299	195746	2.06	0.12
251205433	125281971	7105916	123073740	61268537	273885	1.89	0.15
295089758	147043115	9401172	147097378	71590318	486544	2.13	0.22
350514611	171175887	10582649	165299736	81157551	338651	2.03	0.14
398161674	197871125	12015328	196209949	94192094	368958	2.02	0.13
456745517	227902342	14029353	222981438	106349923	447998	2.05	0.14

Bibliography

[1] Ramesh C. Agarwal and Fred G. Gustavson. *Algorithm and Architecture Aspects of Producing ESSL BLAS on POWER2.*

[2] Gagan Agrawal, Alan Sussman, and Joel Saltz. On efficient runtime support for multiblock and multigrid applications: Regular section analysis. Technical Report CS-TR-3140 and UMIACS-TR-93-92, University of Maryland, Department of Computer Science and UMIACS, October 1993.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, editors. *LAPACK Users' Guide. Second Edition.* SIAM, Philadelphia, 1995.

[4] Jennifer Anderson, Saman Amarsinghe, and Monica Lam. Data and computation transformations for multiprocessors. In *ACM Symposium on Principles and Practice of Parallel Programming*, Jun 1995.

[5] Scott B. Baden, Scott R. Kohn, and Stephen J. Fink. Programming with lparx. Technical Report CS94-377, Department of Computer Science, University of San Diego, 1994.

[6] D Balsara, M Lemke, and D Quinlan. AMR++, a c++ object oriented class library for parallel adaptive refinement fluid dynamics applications. In *American Society of Mechanical Engineers, Winter Annual Meeting, Annaheim, CA*, volume 157, pages 413–433, 1992.

[7] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Workshop on Advances in Languages and Compilers for Parallel Processing*, pages 192–219, August 1990.

[8] Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (Pen)-ultimate tiling? In *INTEGRATION, the VLSI Journal*, volume 17, pages 33–51. 1994.

[9] Steve Carr and R. B. Lehoucq. Compiler blockability of dense matrix factorizations. Technical report, Argonne National Laboratory, Oct 1996.

[10] Steven Carr and Ken Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, November 1994.

[11] Steven Carr and R. B. Lehoucq. A compiler-blockable algorithm for QR decomposition, 1994.

[12] Matthew W. Choptuik, Scott Klasky, and Manish Parashar. Parallel multigrid with the dagh package: Specifications and applications. Draft, September 1995.

[13] Nikos Chrisochoides, Animesh Chatterjee, Rajani Vaidyanathan, and Geoffery Fox. An evaluation of data mapping approaches for parallel multi-block euler solvers. Accepted for publication in Parallel CFD 1994.

[14] Nikos Chrisochoides, Induprakas Kodukula, and Keshav Pingali. Compiler and run-time support for irregular and adaptive applications. In *International Conference on Supercomputing*, Vienna, Austria, Jul 1997. ACM SIGARCH.

[15] Michael Cierniak and Wei Li. Unifying data and control transformations for distributed shared memory machines. In *SIGPLAN 1995 conference on Programming Languages Design and Implementation*, Jun 1995.

[16] Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *ACM Int. Conf. on Supercomputing*. ACM, May 1996.

[17] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In David W. Wall, editor, *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, volume 30(6) of *ACM SIGPLAN Notices*, pages 279–290, New York, NY, USA, June 1995. ACM Press.

[18] Greg Cook, Steve Brandt, Joan Masso, John Shalf, Paul Walker, Matt Choptuik, Mijan Huq, and Manish Parashar. 2d wave equation application.

[19] Jack Dongarra and Robert Schreiber. Automatic blocking of nested loops. Technical Report UT-CS-90-108, Department of Computer Science, University of Tennessee, May 1990.

[20] High Performance Fortran Forum. High performance fortran language specification vertion 1.0, 1993.

[21] High Performance Fortran Forum. High performance fortran language specification, version 2, specification. http://www.crpc.rice.edu/HPFF/hpf2/index.html, October 1996.

[22] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving problems on Concurrent Processors*, volume 1. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[23] Hans Michael Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessor Systems.* PhD thesis, Universitát Bonn, 1990.

[24] Gene Golub and Charles Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 1996.

[25] Y Guan. Unroll-and-jam guided by a linear-algebra-based data-reuse model. Master's thesis, Michigan Technological University, 1995.

[26] S. K. S. Gupta, C.-H. Huang, P. Sadayappan, and R. W. Johnson. Implementing fast Fourier transforms on distributed-memory multiprocessors using data redistributions. *Parallel Processing Letters*, 4(4):477–488, December 1994.

[27] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan-Kaufmann, 1996.

[28] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Unified framework for sparse and dense smpd code generation. Technical Report TR97-1627, Cornell University, 1997.

[29] B. Kumar, C.-H. Huang, R. W. Johnson, and P. Sadayappan. A tensor product formulation of strassen's matrix multiplication algorithm with memory reduction. In V. K. Prasanna, editor, *Proceedings of the 7th International Parallel Processing Symposium*, pages 582–589, Newport Beach, CA, April 1993. IEEE Computer Society Press.

[30] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, California, April 8–11, 1991. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society.

[31] W. Li and K. Pingali. Access Normalization: Loop restructuring for NUMA compilers. *ACM Transactions on Computer Systems*, 1993.

[32] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. In *ACM Transactions on Programming Languages and Systems*, volume 18, pages 424–453. July 1996.

[33] Ketan Mulmuley. *Computation Geometry - An Introduction Through Randomized Algorithms*, chapter 2. Prentice-Hall.

[34] Ketan Mulmuley. *Computational Geometry - An Introduction Through Randomized Algorithms*. Prentice-Hall, Englewood Cliffs, New Jersey, 1994.

[35] Netlib. The basic linear algebra subroutine package.

[36] Manish Parashar and James C. Browne. Distributed dynamic data-structures for parallel adaptive mesh refinement. In *HiPC*, 1995.

[37] Nikos Pitsianis. *The Kronecker Product in Approximation and Fast Transform Generation*. PhD thesis, Cornell University, 1997.

[38] William Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Communications of the ACM*, pages 102–114, August 1992.

[39] William Pugh. Counting solutions to presburger formulas: How and why. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 94.

[40] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *SIGPLAN89 conference on Programming Languages, Design and Implementation*, Jun 1989.

[41] Anne Rogers. *Compiling for Locality of Reference*. PhD thesis, Cornell University, 1991.

[42] Vivek Sarkar. Automatic selection of high order transformations in the IBM ASTI optimizer. Technical Report ADTI-96-004, Application Development Technology Institute, IBM Software Solutions Division, July 1996. Submitted to special issue of IBM Journal of Research and Development.

[43] Robert Schreiber. Personal communication, Nov 1996.

[44] Robert Schreiber. Personal communication, Feb 1997.

[45] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *SIGPLAN 1991 conference on Programming Languages Design and Implementation*, Jun 1991.

[46] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *MICRO 29*, pages 274–286, Silicon Graphics, Mountain View, CA, 1996.

[47] Mike Wolf. *A Loop Transformation Theory and Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford University, August 1992.

[48] M. Wolfe. Iteration space tiling for memory hierarchies. In *Third SIAM Conference on Parallel Processing for Scientific Computing*, December 1987.

[49] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.

[50] H. Zima, H. Bast, and M. gerndt. Superb, a tool for semiautomatic mimd/simd parallelization. In *Parallel Comput.*, pages 1–18, 1986.