LOCALITY ENHANCEMENT OF IMPERFECTLY-NESTED LOOP NESTS

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

Nawaaz Ahmed

August 2000

© Nawaaz Ahmed 2000

ALL RIGHTS RESERVED

LOCALITY ENHANCEMENT OF IMPERFECTLY-NESTED LOOP NESTS

Nawaaz Ahmed, Ph.D.

Cornell University 2000

Most numerical applications using arrays require extensive program transformation in order to perform well on current machine architectures with deep memory hierarchies. These transformations ensure that an execution of the application exploits data-locality and uses the caches more effectively.

The problem of exploiting data-locality is well understood only for a small class of applications – for programs in which all statements are present in the innermost loop of a loop-nest (called perfectly-nested loops). For such programs, statement instances can be mapped to an integer lattice (called the iteration space), and important transformations can be modelled as unimodular transformations of the iteration space. This framework has permitted the systematic application of transformations like loop-permutation, skewing and tiling in order to enhance locality in perfectly-nested loops.

In dealing with programs that do not fall into this category, current compilers resort to ad-hoc techniques to find the right sequence of transformations. For some important benchmarks, no technique is known that will discover the right sequence of

transformations. In my thesis, I propose a technique that extends the framework for perfectly-nested loops to general programs. The key idea is to embed the iteration space of every statement in the program into a special iteration space called the *product space*. The product space can be viewed as a perfectly-nested loop nest, so this embedding generalizes techniques like code sinking and loop fusion that are used in ad hoc ways in current compilers to produce perfectly-nested loops from imperfectly-nested ones. In contrast to these ad hoc techniques however, embeddings are chosen carefully to enhance locality. The product space is then transformed further using unimodular transformations, after which fully permutable loops are tiled, and code is generated. Code can also be generated to emulate block-recursive versions of the original program. I demonstrate the effectiveness of this approach for dense numerical linear algebra benchmarks, relaxation codes, and the tomcatv code from the SPECfp95 benchmark suite.

Biographical Sketch

Nawaaz Ahmed was born on August 1st, 1972 in Tiruchirapalli, India. He graduated from the Indian Institute of Technology, Madras (now Chennai) with a B. Tech. in Computer Science. He joined the Ph.D program in the Department of Computer Science at Cornell University in August 1994. He received an M.S. in Computer Science from Cornell University in 1997.

To Ithaca

This Ithaca has done for me - set me
out upon my way. It cannot then
seem too lean :

my journeys start here, in this calm
destination. I've yet to face
irate Poseidon

and battle the Cyclops. These demons
I bear in my soul, and my soul
will surely

raise them up in front of me. I pray
though the course be long
I am touched

by fine sentiment and lofty thinking
so when old and I moor at last
I haven't lost

the wealth that Ithaca has given me.

Acknowledgements

To Prof. Keshav Pingali, many thanks and much gratitude.
For support and guidance. For keeping me grounded,
for teaching me how to fly.

To my committee members Prof. Bojanczyk and Prof. Zabih
for being available whenever I needed them, for listening
and helping out when they could.

To the Department of Computer Science, for being a haven;
for giving me the freedom to choose what I wanted to do,
for the six year incubation.

To Induprakas Kodukula, Vladimir Kotlyar and Paul Stodghill,
for letting me stand on their shoulders. To Helene Croft,
for smoothening the passage.

To Nikolay Mateev and Vijay Menon – friends, brothers-in-arms.
For toiling with me. For trading dreams and laughter,
nightmares and despair.

To Raghu, for being a wonderful house-mate; for fried eggs,
for pongal and vengaaya kaaram. For putting up with me
for six years. For companionship.

To Sotiris, for sharing and listening, for laughing and commiserating.
To Gihan and Swati, for fellowship and empathy.
For berry-picking and chicken soup.

To the Dance department. For being a second home. For concerts
and studios. For the Proscenium stage. For setting me free.
For friends and companions.

To Prof. Joyce Morgenroth, for helping add wings to my feet.
For support and encouragement, for being my minor advisor.
For Tracings and center stage.

To Jim, Jumay, Janice, Joyce and Byron. For holding my hand,
for patting my back. For plies and tendus. For helping me
find myself. For Dance.

To my parents, my brother and my sister for their love and support
steady across continents and oceans. For believing I could do it.
For letting me live my dreams.

Table of Contents

List of Figures

Chapter 1

Introduction

The memory systems of computers are organized as a hierarchy in which the latency of memory accesses increases by roughly an order of magnitude from one level of the hierarchy to the next. In such architectures, a program will run well only if it exhibits enough locality of reference so that most of its data accesses are satisfied by the faster levels of the memory hierarchy. Unfortunately, most programs produced by straight-forward coding of algorithms do not exhibit sufficient locality of reference. The numerical linear algebra community has addressed this problem by writing libraries of carefully hand-crafted programs such as the Basic Linear Algebra Subroutines (BLAS) [29] and LAPACK [2] for algorithms of interest to their community. However, these libraries are useful only when linear systems solvers or eigensolvers are needed, so they cannot be used when explicit methods are used to solve partial differential equations (pde's), for example.

The restructuring compiler community has explored a more *general-purpose* approach in which program locality is enhanced through restructuring by a compiler which does not have any knowledge of the algorithms being implemented by these programs. In principle, such technology can be brought to bear on any program

```
for t = 1,T
    for i1 = 2,N-1
        for j1 = 2,N-1
S1:         L(i1,j1) = (A(i1,j1+1) + A(i1,j1-1)
                      + A(i1+1,j1) + A(i1-1,j1)) / 4
        end
    end
    for i2 = 2,N-1
        for j2 = 2,N-1
S2:         A(i2,j2) = L(i2,j2)
        end
    end
end
```

Figure 1.1: Jacobi : Original Code

without restriction to problem domain. This technology transforms programs using a set of loop transformations like interchange, reversal, skewing, fission, fusion etc.

Consider the code fragment shown in Figure 1.1. It is typical of code in *pde* solvers that use explicit methods. These are called relaxation codes in the compiler literature. They contain an outer loop that counts time-steps; in each time-step, a smoothing operation (stencil computation) is performed on arrays that represent approximations to the solution to the *pde*. As can be seen, each statement walks over both arrays `A` and `L`. This results in bringing these arrays twice through cache for each iteration of the outer `t` loop. Clearly both the statements are touching the same arrays locations and there is reuse between statements `S1` and `S2`. Furthermore, each iteration of the `t`-loop itself touches the same data, and therefore the `t`-loop carries reuse too. If the arrays do not fit into cache, these reuses will not be exploited. It is possible to exploit the reuse between statements for a given iteration of `t` if the code is rewritten as in Figure 1.2(a). This code can be obtained by peeling away the first iterations of the `i1`,`j1` loops and the last iterations of the `i2`,`j2` loops and then fusing the remaining `i1` and `i2` loops as well as `j1` and `j2` loops. Reuse between

```
for t = 1,T
    for j1 = 2, N-1
        L(2, j1) = (A(2,j1+1) + A(2,j1-1)
                   + A(4,j1) + A(1,j1)) / 4
    end
    for i = 3,N-1
        L(i, 2) = (A(i,3) + A(i,1)
                  + A(i+1,2) + A(i-1,2)) / 4
        for j = 3,N-1
            L(i,j) = (A(i,j+1) + A(i,j-1)
                    + A(i+1,j1) + A(i-1,j1)) / 4
            A(i-1,j-1) = L(i-1,j-1)
        end
        A(i-1,N-1) = L(i-1,N-1)
    end
    for j2 = 2,N-1
        A(N-1,j2) = L(N-1,j2)
    end
end
```

Figure 1.2: Jacobi : Optimized for locality

different iterations of the `t`-loop can be exploited by skewing the resulting `i` and `j` loops by `2*t` and then tiling all three loops.

Though the technology for each of these steps is implemented in many commercial compilers, each code may require a different sequence of steps, and finding the best sequence is non-trivial. Most techniques heuristically search for a good sequence of transformations [34, 31, 6, 35]. We do not know any commercial compiler that finds the right sequence for the Jacobi example.

An alternative to searching for a good sequence of transformations has been developed for perfectly-nested loop nests. A perfectly-nested loop nest is a set of loops in which all assignment statements are contained in the innermost loop; the matrix multiplication kernel shown in Figure 1.3(a) is an example of such a loop nest.

For perfectly-nested loop nests, polyhedral methods can be used to *synthesize*

```
for i = 1,N
    for j = 1,N
        for k = 1,N
            c(i,j) = c(i,j) + a(i,k) * b(k,j)
        end
    end
end
```

(a) Matrix Multiplication Kernel

```
//tile counter loops
for t1 = 1, N, B
    for t2 = 1, N, B
        for t3 = 1, N, B
            //iterations within a tile
            for j = t1, min(t1+B-1,N)
                for k = t2, min(t2+B-1,N)
                    for i = t3, min(t3+B-1,N)
                        c(i,j) = c(i,j) + a(i,k) * b(k,j)
                    end
                end
            end
        end
    end
end
```

(b) Optimized Matrix Multiplication Kernel

Figure 1.3: Optimizing Matrix Multiplication

sequences of linear loop transformations (permutation, skewing, reversal and scaling) for enhancing and locality [1, 4, 5, 33, 23, 28, 3]. The key idea is to model the iterations of the loop nest as points in an integer lattice, and to model linear loop transformations as nonsingular matrices mapping one lattice to another. A sequence of loop transformations is modeled by the product of matrices representing the individual transformations; since the set of nonsingular matrices is closed under matrix multiplication, this means that a sequence of linear loop transformations is also represented by a nonsingular matrix. The problem of finding an optimal sequence of linear loop transformations is thus reduced to the problem of finding an integer matrix that satisfies some desired property. This formulation has permitted the full machinery of matrix methods and lattice theory to be applied to the loop transformation synthesis problem for perfectly-nested loops. Highlights of this technology are the following. On the matrix multiplication example in Figure 1.3(a), a compiler might permute the loops to obtain the j-k-i ordering (which is good for the spatial locality of the loop nest assuming the matrices are stored in Fortran order as the accesses to c(i,j) and a(i,k) will be unit-stride), and then, since all three loops exhibit temporal reuse, the compiler might tile the loops as shown in Figure 1.3(b). Of course, these transformations are not always legal or beneficial. The legality and benefit of these transformations have been extensively studied [4, 32, 35, 22, 30]. This technology has been incorporated into production compilers such as the SGI MIPSPro compiler, enabling these compilers to produce good code for perfectly-nested loop nests.

In real programs though, many loop nests are *imperfectly-nested* (that is, one or more assignment statements are contained in some but not all of the loops of the loop nest). The Jacobi example in Figure 1.1 is imperfectly-nested as are other relaxation codes. Cholesky, LU and QR factorizations [14] also contain imperfectly-nested loop nests. An entire program or subroutine, which usually is a sequence of perfectly- or imperfectly-nested loop nests, can itself be considered to be imperfectly nested.

A number of approaches have been proposed for enhancing locality of reference in imperfectly-nested code segments. The simplest approach is to transform each maximal perfectly-nested loop nest separately. In the Jacobi example in Figure 1.1, the `i1` and `j1` loops, the `i2` and `j2` loops, and the `t` loop by itself together form three maximal perfectly-nested loop nests. These can be transformed using techniques for perfectly-nested loop nests but this will not result in the optimized code in Figure 1.2(b).

A more aggressive approach is to (i) convert an imperfectly-nested loop nest into a perfectly-nested loop nest if possible by applying transformations like *code sinking*, *loop fusion* and *loop fission* [36], and then (ii) use locality enhancement techniques for the resulting maximal perfectly-nested loops. In general, there are many ways to do this conversion, and the performance of the resulting code may depend critically on how this conversion is done. For example, certain orders of applying code sinking, fission and fusion might lead to code that cannot be tiled, while other orders could result in tilable code [19].

The above approach is further complicated by the fact that transformations like loop fission and fusion themselves are useful in improving data locality of loop nests. Loop fusion has been shown to improve data locality by addressing inter-loop nest reuse. Loop fission, on the other hand, can decrease the number of cache misses in some cases by allowing less data to pollute the cache.

To avoid these difficulties, we would like to generalize techniques developed for perfectly-nested loops to imperfectly-nested loop nests. This requires us to solve three problems. First, how do we represent imperfectly-nested loop nests and specify transformations for them? Second, how do we ensure legality of these transformations? And finally, how do we describe and find transformations that enhance locality?

One way to solve the first problem is to use techniques developed by the systolic array community for scheduling statements in loop nests on systolic arrays. These schedules specify mappings from statement instances to processor/time axes; these

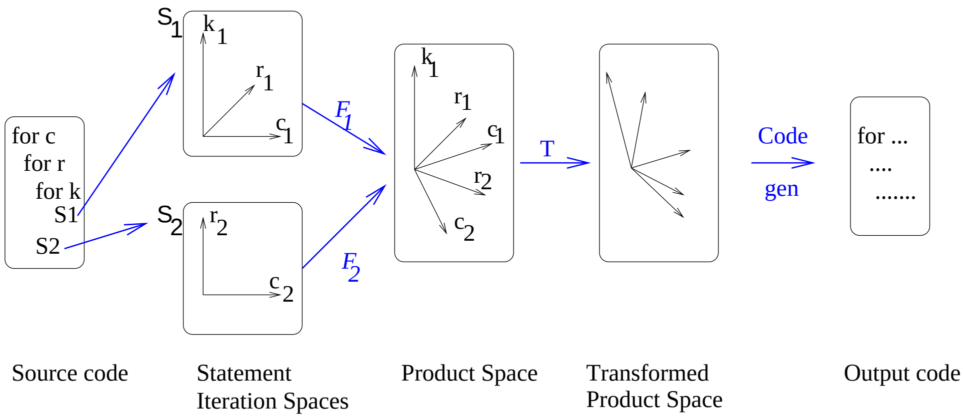


Figure 1.4: Locality Enhancement of Imperfectly-nested Loop Nests

mappings are usually restricted to be affine functions of loop variables [20]. It is straight-forward to interpret these schedules or mappings as loop transformations in which each assignment statement in a loop nest is mapped by a possibly different linear (or pseudo-linear) function to a target iteration space. These techniques were extended by Feautrier in his theory of *schedules* in multi-dimensional time [11, 12]; a related approach is Kelly and Pugh's *mappings* [16] and Lim and Lam's *affine transforms* [24]. The second problem—finding legal transformations—can be be solved by an application of *Farkas' lemma* (Section 2.4.1). Feautrier and Lim et al address the third problem of describing and finding desirable transformations for parallelism. Feautrier searches for parallel schedules of minimum length using parametric integer programming and Lim et al identify all degrees of parallelism in a program. Kelly and Pugh advocate searching the space of legal transformations using external cost models that examine the mappings produced.

In this thesis, we propose an approach for locality enhancement of imperfectly-nested loops that generalizes the approach used for perfectly-nested loops. Our strategy is shown in Figure 1.4. The iteration space of each statement is embedded by means of an affine mapping in a space we call the *product space* by means of an affine mapping. The product space is chosen to be large enough so that there is a legal way of embedding statements in it with a 1-1 mapping. *These embeddings are chosen so as to maximize reuse in the program.* Embeddings generalize transformations like code-sinking, loop fusion, and loop distribution that convert imperfectly-nested loop nests into perfectly-nested ones, and are specified by affine embedding functions F_i as shown in Figure 1.4. The resulting space can then be further transformed for locality enhancement using the well understood techniques for perfectly-nested loops like height-reduction [23] and tiling.

The rest of this thesis is organized as follows. In Chapter 2, I describe the problem of locality enhancement and develop the product space framework to handle it. Chapter 3 develops algorithms that use this framework to produce embeddings that enable tiling and enhance reuse. The results of applying this technique to various important benchmarks is described in Chapter 3.5. Chapter 4 further extends the framework to generate block-recursive versions of these codes automatically. Experimental results from applying this extension is presented in Chapter 4.4. Finally, Chapter 5 summarizes the contributions of this dissertation and discusses open problems.

Chapter 2

A Framework for Locality Enhancement

2.1 Modeling Program Execution

A program is assumed to consist of statements contained in perfectly- and imperfectly-nested loop nests. All loop bounds and array access functions are assumed to be affine functions of surrounding loop indices. We will use S_1, S_2, $\ldots$, S_n to name the statements in the program in syntactic order. A *dynamic instance* of a statement S_k refers to a particular execution of the statement for a given value of index variables i_k of the loops surrounding it, and is represented by $S_k(i_k)$.

Executing a program imposes a total order on the dynamic statement instances of the program – this is the order in which the statement instances are executed. Consider for example the two programs shown in Figure 2.1. Both the programs have the same set of dynamic statement instances (shown below for $N = 5$) :

$$\{S_1(1), S_1(2), SS_1(3), S_1(4), S_1(5), S_2(1), S_2(2), S_2(3), S_2(4), S_2(5)\}$$

```
for i1 = 1, N
S1: x(i1) = a(i1)
end

for i2 = 1, N
S2: X(i2) = x(i2) + a(i2)
end
```

(a) Example 1

```
for i = 1, N
S1: x(i) = a(i)
S2: X(i) = x(i) + a(i)
end
```

(b) Example 2

Figure 2.1: Code Fragments

Executing the two programs orders this set of statement instances differently. The total order imposed by executing the first program is

$$S_1(1), S_1(2), S_1(3), S_1(4), S_1(5), S_2(1), S_2(2), S_2(3), S_2(4), S_2(5)$$

while the second program imposes the following total order :

$$S_1(1), S_2(1), S_1(2), S_2(2), S_1(3), S_2(3), S_1(4), S_2(4), S_1(5), S_2(5)$$

In order to reason about the effect of executing a program we need only to know the set of dynamic statement instances and the total order in which they are executed. This allows us to abstract out the actual code written in terms of a sequence of loops.

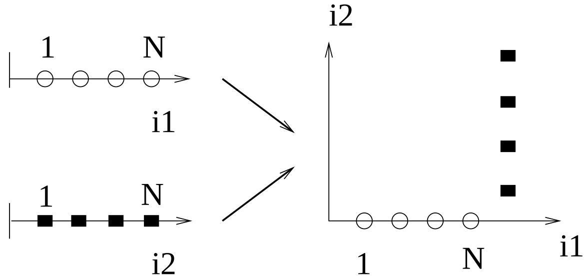
We can model the execution order of a set of dynamic statement instances by defining a *Program Iteration Space*, as follows:

1. Let $\mathcal{P}$ be a p-dimensional Cartesian space for some p.

2. Embed all dynamic statement instances $S_k(i_k)$ into $\mathcal{P}$ using embedding functions F_k which satisfy the following constraints:

   (a) Each F_k must be one-to-one[1].

[1]Note that instances of *different* statements may get mapped to a single point of the program iteration space.

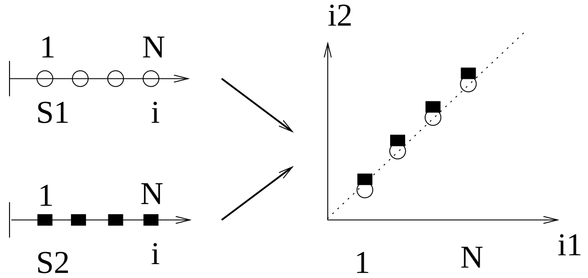


(a) Space

$$F_1(i_1) = \begin{bmatrix} i_1 \\ 0 \end{bmatrix} \quad F_2(i_2) = \begin{bmatrix} N+1 \\ i_2 \end{bmatrix}$$

(b) Embedding Functions

Figure 2.2: Space and Embedding Functions for Example 1



(a) Space

$$F_1(i_1) = \begin{bmatrix} i_1 \\ i_1 \end{bmatrix} \quad F_2(i_2) = \begin{bmatrix} i_2 \\ i_2 \end{bmatrix}$$

(b) Embedding Functions

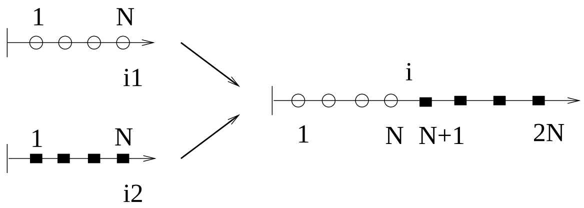
Figure 2.3: Space and Embedding Functions for Example 2

(b) If the points in space $\mathcal{P}$ are traversed in lexicographic order, and all statement instances mapped to a point are executed in original program order when that point is visited, the program execution order is reproduced.

An execution order can thus be modeled by the pair $(\mathcal{P}, \mathcal{F} = F_1, F_2, \ldots, F_n\})$.

For example, the execution order of the code shown in Figure 2.1(a) can be represented by mapping the statements to a 2-dimensional space as shown in Figure 2.2(a). The embedding functions for the two statements are shown in Figure 2.2(b). Similarly, the execution order of Example (2) can be modeled as shown in Figure 2.3.

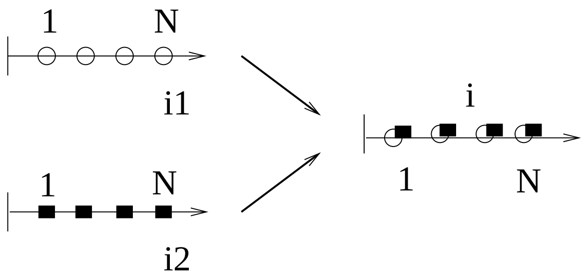
Of course, these spaces and embedding functions are not the only way to model the execution order of these code fragments. Models that use a 1-dimensional space instead of a 2-dimensional space are shown in Figure 2.4 and Figure 2.5.



$$F_1(i_1) = \begin{bmatrix} i_1 \end{bmatrix} \quad F_2(i_2) = \begin{bmatrix} N + i_2 \end{bmatrix}$$

(a) Space (b) Embedding Functions

Figure 2.4: Space and 1D Embedding Functions for Example 1



$$F_1(i_1) = \begin{bmatrix} i_1 \end{bmatrix} \quad F_2(i_2) = \begin{bmatrix} i_2 \end{bmatrix}$$

(a) Space (b) Embedding Functions

Figure 2.5: Space and 1D Embedding Functions for Example 2

2.1.1 Optimizing Programs

Consider the two programs shown in Figure 2.1. Executing the two programs has the same result on the data –

$$\forall_{i=1}^{5} x(i) = 2 * a(i)$$

Hence the two total orders are effectively equivalent as far as the semantics of the two programs is concerned. On the other hand, these two versions may perform very differently. If the arrays `x` and `a` do not fit in cache, the second version will incur fewer cache misses and will have fewer load and store instructions and is therefore likely to perform better.

One way of optimizing the first program is to transform its execution order (which we will call the *original execution order* shown in Figure 2.2) into the execution order corresponding to the second program (Figure 2.3) and generating code that can emulate that transformed execution order (in this case the program in Example 2).

The problem of optimizing a program can thus be reduced to the problem of searching for an execution order $(\mathcal{P}, \mathcal{F})$ that is somehow "better" than the original execution order and still preserves the semantics of the original program.

2.1.2 Legality of Execution Orders

The original execution order of a program can be transformed legally into an order $(\mathcal{P}, \tilde{\mathcal{F}})$ if the latter preserves the semantics of the program. Dependence analysis states that semantics of a program will be preserved if all dependences are preserved under the transformation.

Dependences

A dependence exists from instance i_s of statement `Ss` to instance i_d of statement `Sd` if the following conditions are satisfied.

1. *Loop bounds*: Both source and destination statement instances lie within the corresponding iteration space bounds. Since the iteration space bounds are affine expressions of index variables, we can represent these constraints as $B_s * i_s + b_s \geq 0$ and $B_d * i_d + b_d \geq 0$ for suitable matrices B_s, B_d and vectors b_s, b_d.

2. *Same array location*: Both statement instances reference the same array location and at least one of them writes to that location. Since the array references are assumed to be affine expressions of the loop variables, these references can be written as $A_s * i_s + a_s$ and $A_d * i_d + a_d$. Hence the existence of a dependence requires that $A_s * i_s + a_s = A_d * i_d + a_d$.

```
for c = 1,M
    for r = 1,N
        for k = 1,r-1
S1:         B(r,c) = B(r,c) - L(r,k) * B(k,c)
        end
S2:     B(r,c) = B(r,c)/L(r,r)
    end
end
```

Figure 2.6: Triangular Solve with Multiple Right-hand Sides

3. *Precedence order*: Instance i_s of statement `Ss` occurs before instance i_d of statement `Sd` in program execution order. If $common_{sd}$ is a function that returns the loop index variables of the loops common to both i_s and i_d, this condition can be written as $common_{sd}(i_d) \succeq common_{sd}(i_s)$ if `Sd` follows `Ss` syntactically or $common_{sd}(i_d) \succ common_{sd}(i_d)$ if it does not, where $\succ$ is the lexicographic ordering relation.

   This condition can be translated into a disjunction of matrix inequalities of the form $X_s * i_s - X_d * i_d + x \geq 0$.

If we express the dependence constraints as a disjunction of conjunctions, each term in the resulting disjunction can be represented as a matrix inequality of the following form.

$$D \begin{bmatrix} i_s \\ i_d \end{bmatrix} + d = \begin{bmatrix} B_s & 0 \\ 0 & B_d \\ A_s & -A_d \\ -A_s & A_d \\ X_s & -X_d \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} + \begin{bmatrix} b_s \\ b_d \\ a_s - a_d \\ a_d - a_s \\ x \end{bmatrix} \geq 0$$

Each such matrix inequality will be called a *dependence class*, and will be denoted by $\mathcal{D}$ with an appropriate subscript. For the code in Figure 2.6, it is easy to show that there are two dependence classes[2]. The first dependence class $\mathcal{D}_1$ arises because statement `S1` writes to a location `B(r,c)` which is then read by statement `S2`; similarly, the second dependence class $\mathcal{D}_2$ arises because statement `S2` writes to location `B(r,c)` which is then read by reference `B(k,c)` in statement `S1`.

$$
\begin{array}{rrcrcrcrcrcl}
\mathcal{D}_1 : & \mathtt{M} & \geq & c_1 & \geq & 1 & \mathtt{M} & \geq & c_2 & \geq & 1 \\
& \mathtt{N} & \geq & r_1 & \geq & 1 & \mathtt{N} & \geq & r_2 & \geq & 1 \\
& r_1 - 1 & \geq & k_1 & \geq & 1 \\
& r_1 & = & r_2 \\
& c_1 & = & c_2
\end{array}
$$

$$
\begin{array}{rrcrcrcrcrcl}
\mathcal{D}_2 : & \mathtt{M} & \geq & c_1 & \geq & 1 & \mathtt{M} & \geq & c_2 & \geq & 1 \\
& \mathtt{N} & \geq & r_1 & \geq & 1 & \mathtt{N} & \geq & r_2 & \geq & 1 \\
& r_1 - 1 & \geq & k_1 & \geq & 1 \\
& k_1 & = & r_2 \\
& c_1 & = & c_2
\end{array}
$$

Legality

Let $(\mathcal{P}, \mathcal{F} = \{F_1, F_2, \ldots F_n\})$ be an execution order for a program. We will say that these embedding functions are *legal* if for every (i_s, i_d) in every dependence class $\mathcal{D}$, the point that i_s is mapped to in the program iteration space is lexicographically less than the point that i_d is mapped to. Since we are traversing the program iteration space lexicographically, this ensures that the source of the dependence (the point $F_s(i_s)$) is visited before the destination of the dependence (the point $F_d(i_d)$). If we

[2]There are other dependences, but they are redundant.

execute all statement instances mapped to the point when the point is visited [3], this will ensure that no dependence is violated.

For future reference, we define this formally.

Definition 1 *Let $\mathcal{F} = \{F_1, F_2, \dots F_n\}$ be embedding functions that embed the statement iteration spaces of a program into a space $\mathcal{P}$. These embedding functions are said to be* legal *if for every dependence class $\mathcal{D}$ of the program,*

$$\forall (i_s, i_d) \in \mathcal{D} \quad F_d(i_d) \succeq F_s(i_s) \tag{2.1}$$

We will refer to the vector $F_d(i_d) - F_s(i_s)$ as the *difference vector* for $(i_s, i_d) \in \mathcal{D}$.

2.2 Locality Enhancement

In order to use the approach to optimization described in Section 2.1.1, there must be a way of comparing execution orders. The most accurate measure of the "goodness" of an execution order is, of course, the time it takes to execute it. Clearly we want the execution order with the shortest execution time. While this metric (i.e. execution time) can be used to select between a few choices of execution orders, it is not practical to use it when the number of choices is large. Also, this metric is machine dependent and as machine architectures grow more complex, it becomes impossible to model it accurately.

One metric to measure the effectiveness of an execution order with respect to its memory hierarchy performance is to count the number of cache misses caused by that execution order. While this metric can be measured fairly accurately during execution with the help of special-purpose hardware counters, it has proven

[3] If there are more than one statement instances mapped to a point, then execute them in original program order.

to be hard to model cache misses accurately. The number of cache misses is a function of the size of the cache, the cache-line size, the cache associativity, the cache-replacement policy, the data allocation policy of the compiler or the runtime system and is difficult to model except in the simplest cases (which do not reflect current processor architectures). This metric is further complicated by the fact that modern processors have more than one level of cache and each level might incur a different number of cache misses. Further, the correlation between number of cache misses and the time spent servicing them is not straight-forward : the effective time spent in handling cache misses depends on the level of the cache, the number of outstanding misses, whether the processor is required to block during a cache miss, the number of simultaneous loads/stores that the processor can issue etc. and is too complicated to model accurately.

Given the complexity inherent in these two metrics, we would like a metric that is architecture independent and depends only on the execution order in consideration. Caches are useful only if the program exhibits reuse – that is, the same memory location is accessed by different dynamic statement instances. Further even if the program exhibits reuse, the reuse might not be exploited by an execution order. In order for a reuse between two statement instances $s_1(i_1)$,$s_2(i_2)$ to be exploited by an execution order, the common data accessed by $s_1(i_1)$ must reside in the cache when $s_2(i_2)$ is executed. This implies that the data touched by the intervening statement instances must not evict the required data before $s_2(i_2)$ is executed. In the general case, this cannot be guaranteed unless the required data can be placed in a register for the duration between the two statement instances. Since this might not always be possible, the alternate option is to reduce the likelihood of the common data being evicted before it is reused. One way to achieve this is to reduce the *reuse distance* between statement instances exhibiting reuse – reuse distance is the number of intervening statement instances in the execution order.

Consider the data access patterns of the examples in Figure 2.1. Statement instances `S1(i)` and `S2(i)` *exhibit data reuse* because they touch the same memory location `x(i)`. The number of statement instances executed between them is the reuse distance. In the first example, the reuse distance is $N - 1$. If N is larger than the cache-size then the reuse will not be exploited since each statement instance brings in additional data into the cache. On the other hand, the reuse distance in the second example is 0 – there are no intervening statement instances between `S1(i)` and `S2(i)` and hence this reuse is will be exploited.

If we represent the execution order by $(\mathcal{P}, \mathcal{F})$, the reuse distance between `S1(i)` and `S2(i)` is proportional to the number of points in $\mathcal{P}$ with statements mapped to them that lie lexicographically between the points to which `S1(i)` and `S2(i)` are mapped. This is because of the initial one-to-one mapping requirement—there are utmost a constant number of statement instances (one from each statement) mapped to each point in the space.

In light of the above discussion, our strategy for optimizing programs for efficient memory hierarchy performance is to search for execution orders $(\mathcal{P}, \mathcal{F})$ that reduce reuse distances between statement instances while preserving the correctness of the programs.

2.2.1 Representing Reuses

Formally, a reuse exists from instance i_s of statement S_s (the source of the reuse) to instance i_d of statement S_d (the destination) if the following conditions are satisfied:

1. *Loop bounds*: Both source and destination statement instances lie within the corresponding iteration space bounds. Since the iteration space bounds are affine expressions of index variables, we can represent these constraints as $B_s * i_s + b_s \geq 0$ and $B_d * i_d + b_d \geq 0$ for suitable matrices B_s, B_d and vectors b_s, b_d.

2. *Same array location*: Both statement instances reference the same memory location. If we restrict memory references to array references, these references can be written as $A_s * i_s + a_s$ and $A_d * i_d + a_d$. Hence the existence of a reuse requires that $A_s * i_s + a_s = A_d * i_d + a_d$.

3. *Precedence order*: Instance i_s of statement S_s occurs before instance i_d of statement S_d in program execution order. If $common_{sd}$ is a function that returns the loop index variables of the loops common to both i_s and i_d, this condition can be written as $common_{sd}(i_d) \succeq common_{sd}(i_s)$ if S_d follows S_s syntactically or $common_{sd}(i_d) \succ common_{sd}(i_d)$ if it does not, where $\succ$ is the lexicographic ordering relation. This condition can be translated into a disjunction of matrix inequalities of the form $X_s * i_s - X_d * i_d + x \geq 0$.

If we express the reuse constraints as a disjunction of conjunctions, each term in the resulting disjunction can be represented as a matrix inequality of the following form.

$$R \begin{bmatrix} i_s \\ i_d \end{bmatrix} + r = \begin{bmatrix} B_s & 0 \\ 0 & B_d \\ A_s & -A_d \\ -A_s & A_d \\ X_s & -X_d \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} + \begin{bmatrix} b_s \\ b_d \\ a_s - a_d \\ a_d - a_s \\ x \end{bmatrix} \geq 0$$

Each such matrix inequality will be called a *reuse class*, and will be denoted by $\mathcal{R}$ with an appropriate subscript.

The above definition applies to *temporal* reuses where the same array location is accessed by the source and the destination. If the cache line contains more than one array element, then we can also consider *spatial reuse* where the same cache line is accessed by the source and the destination of the reuse. Spatial reuse depends on the storage order of the array.

The conditions for spatial reuse are similar to the ones for temporal reuse, the only difference being that instead of requiring both statement instances to touch the same array location, we require that the two statement instances touch *nearby array locations that fit in the same cache line*. We can represent this condition as a matrix inequality by requiring the first[4] row of $A_d * i_d + a_d - A_s * i_s - a_s$ to lie between 1 and $c - 1$, where c is the number of array elements that fit into a single cache line, instead of being equal to 0.

For the example in Figure 2.1(a), the temporal reuse existing between $S_1(i_1)$ and $S_2(i_2)$ can be represented by the reuse class $\mathcal{R}_1 = \{(i_1, i_2) : 1 \leq i_1, i_2 \leq N, i_1 = i_2\}$. It is straightforward to represent these inequalities as matrix inequalities. There also exists spatial reuse between these two statements. For a cache line containing 4 array elements, the spatial reuse can be represented by the reuse class $\mathcal{R}_2 = \{(i_1, i_2) : 1 \leq i_1, i_2 \leq N, 1 \leq i_2 - i_1 \leq 3\}$.

2.2.2 Examples of Reuse Distance Reduction

The distance between statement instances that exhibit reuse can be reduced by a number of loop transformations as shown in the following examples. These transfor-

[4]for Fortran storage order.

mations may not always be legal – the transformed code must preserve dependences.

Fusion

If reuse exists between statement instances of two loops, the reuse distance can sometimes be reduced by fusing the two loops appropriately so that the instances exhibiting reuse are executed in the same iteration.

```
for i1 = 1..n
S1: x(i1) = b(i1)
end

for i2 = 1..n
S2: y(i2) = b(i2+1)
end
```

(a) Original Code

```
S1: x(1) = b(1)

for i = 2..n
S1: x(i)   = b(i)
S2: y(i-1) = b(i)
end

S2: y(n) = b(n+1)
```

(b) Fused Code

Figure 2.7: Fusion to reduce reuse distance

In the code in Figure 2.7(a), there is reuse from statement instance `S1(i)`to`S2(i` $-$ `1)` for $1 \leq i \leq n$ since these statement instances access the same array location $b(i)$. The reuse distance between these statement instances is $n-1$. This can be reduced to 0 by mapping the statement instances to a 1-dimensional space as follows –

$$F_1(i_1) = \left[\begin{array}{c} i_1 \end{array}\right] \quad F_2(i_2) = \left[\begin{array}{c} i_2 + 11 \end{array}\right]$$

The resulting code, shown in Figure 2.7(b), exploits the reuse of array `b`.

Loop Permutations

Reuse distances can be reduced by loop-permutations as shown in the example in Figure 2.8(a). The array location x(i) is accessed in each iteration of the outermost `t`-loop. The reuse distance between the reuse statement instances is n. This distance can be reduced to 1 by permuting the `t`-loop so that it is innermost (Figure 2.8(b)).

```
for t = 1..m
    for i = 1..n
S1:     x(i) = x(i) + i
    end
end
```

(a) Original Code

```
for i = 1..n
    for t = 1..m
S1:     x(i) = x(i) + i
    end
end
```

(b) Permuted Code

Figure 2.8: Permutation to reduce reuse distance

This transformation corresponds to the following embedding –

$$F_1(\begin{bmatrix} t \\ i \end{bmatrix}) = \begin{bmatrix} i \\ t \end{bmatrix}$$

Tiling

As shown in the previous example, reuse distances corresponding to certain reuses can be reduced by permuting the loop that contributes most to the reuse distance to an innermost position. If more than one loop affects the reuse distance or if there are multiple reuses in the code, this might not always be possible. In this case, it might be possible to reduce the reuse distance to a small enough value so that the data accessed still remains in the cache. This can be achieved by tiling.

In matrix multiplication (Figure 2.9(a)), all three array references exhibit reuse. The same array location `c(i,j)` is accessed in every `k`-iteration (reuse distance is n^2); the array location `a(i,k)` is accessed in every `j`-iteration (reuse distance is 1) and the array location `b(k,j)` is accessed in every `i`-iteration (reuse distance is n). Each of these reuses can be made to have a reuse distance of 1 by permuting the appropriate loop innermost. But since all three loops cannot be move innermost, the solution is to tile the three loops. This corresponds to embedding the code in a 6-dimensional space as follows –

```
for i = 1,N
    for j = 1,N
        for k = 1,N
            c(i,j) = c(i,j) + a(i,k) * b(k,j)
        end
    end
end
```

(a) Matrix Multiplication Kernel

```
//tile counter loops
for t1 = 1,N, B
    for t2 = 1, N, B
        for t3 = 1, N, B
            //iterations within a tile
            for j = t1, min(t1+B-1,N)
                for k = t2, min(t2+B-1,N)
                    for i = t3, min(t3+B-1,N)
                        c(i,j) = c(i,j) + a(i,k) * b(k,j)
                    end
                end
            end
        end
    end
end
```

(b) Optimized Matrix Multiplication Kernel

Figure 2.9: Optimizing Matrix Multiplication

$$F_1(\begin{bmatrix} k \\ i \\ j \end{bmatrix}) = \begin{bmatrix} k/B \\ i/B \\ j/B \\ k\%B \\ i\%B \\ j\%B \end{bmatrix}$$

Under this embedding scheme ((Figure 2.9(b)), by choosing an appropriate value of B (called the tile size) we can limit the distance between the majority of accesses to the same array location to be less than B^2 for all three references. This can result in the data reused remaining in the cache between reuses.

2.2.3 Minimizing Reuse Distances

Let $\mathcal{F} = \{F_1, F_2, \ldots F_n\}$ be embedding functions that embed the statement iteration spaces of a program into a space $\mathcal{P}$, and let $\mathcal{R}$ be any reuse class for that program. Consider any reuse pair $(i_s, i_d) \in \mathcal{R}$. Let $Distance(i_s, i_d)$ be the number of points in the space $\mathcal{P}$ with statements mapped to them that lie lexicographically between $F_s(i_s)$ and $F_d(i_d)$. As we saw in Section 2.2, the reuse distance between $S_s(i_s)$ and $S_d(i_d)$ is proportional to $Distance(i_s, i_d)$. We define $ReuseDistances(\mathcal{P}, \mathcal{F})$ to be the vector of $Distance(i_s, i_d)$ for all reuse pairs (i_s, i_d) in the program under the execution order $(\mathcal{P}, \mathcal{F})$.

Our goal for locality enhancement is to find legal execution orders $(\mathcal{P}, \mathcal{F}_{opt})$ that minimize

$$\|ReuseDistances(\mathcal{P}, \mathcal{F})\|_X$$

for some suitable norm $\| \cdot \|_X$.

2.3 Product Space

The search space for optimal execution orders as described in the previous section is too huge to be practical. In this section, we define a special space called the *product space* and restrict the set of embedding functions we will consider. The corresponding restricted execution orders are still powerful enough to capture most of the transformations we are interested in. As our running example we will use the code in Figure 2.10 which implements relaxation in one dimension.

```
for t = 1, M
   for j1 = 2, N-1
S1:     L(j1) = A(j1-1) + A(j1+1)
   end
   for j2 = 2, N-1
S2:     A(j2) = L(j2)
   end
end
```

Figure 2.10: 1-D Relaxation Code

2.3.1 Statement Iteration Spaces

We associate a distinct iteration space with each statement in the loop nest, as described in Definition 2.

Definition 2 *Each statement in a loop nest has a* statement iteration space *whose dimension is equal to the number of loops that surround that statement.*

We will use `S1`, `S2`, ... , `Sn` to name the statements in the loop nest in syntactic order. The corresponding statement iteration spaces will be named $\mathcal{S}_1$, $\mathcal{S}_2$, ... , $\mathcal{S}_n$. In Figure 2.10, the iteration space $\mathcal{S}_1$ of statement `S1` is the two-dimensional space $t_1 \times j_1$, while the iteration space $\mathcal{S}_2$ of `S2` is a two-dimensional space $t_2 \times j_2$.

The bounds on statement iteration spaces can be specified by integer linear inequalities. For our running example, these bounds are the following:

$$
\begin{array}{rcccc rcccc}
\mathcal{S}_1 : \mathtt{M} & \geq & t_1 & \geq & 1 & \mathcal{S}_2 : \quad \mathtt{M} & \geq & t_2 & \geq & 1 \\
\mathtt{N}-\mathtt{1} & \geq & j_1 & \geq & 2 & \mathtt{N}-\mathtt{1} & \geq & j_2 & \geq & 2
\end{array}
$$

An *instance* of a statement is a point within that statement's iteration space.

2.3.2 Product Spaces and Embedding Functions

The *product space* for a loop nest is the Cartesian product of the individual statement iteration spaces of the statements within that loop nest. The order in which this product is formed is the syntactic order in which the statements appear in the loop nest. For our running example, the product space is the four-dimensional space $t_1 \times j_1 \times t_2 \times j_2$.

The relationship between statement iteration spaces and the product space is specified by projection and embedding functions. Suppose $\mathcal{P} = \mathcal{S}_1 \times \mathcal{S}_2 ... \times \mathcal{S}_n$. Projection functions $\pi_i : \mathcal{P} \rightarrow \mathcal{S}_i$ extract the individual statement iteration space components of a point in the product space, and are obviously linear functions. For our running example, $\pi_1 = \begin{bmatrix} I_{2\times 2} & 0 \end{bmatrix}$ and $\pi_2 = \begin{bmatrix} 0 & I_{2\times 2} \end{bmatrix}$.

An embedding function F_i on the other hand maps a point in statement iteration space $\mathcal{S}_i$ to a point in the product space. Unlike projection functions, embedding functions can be chosen in many ways. In our framework, we will consider only those embedding functions $F_i : \mathcal{S}_i \rightarrow \mathcal{P}$ that satisfy the following conditions.

Definition 3 *Let* `Si` *be a statement whose statement iteration space is* $\mathcal{S}_i$*, and let* $\mathcal{P}$ *be the product space. An* embedding function $F_i : \mathcal{S}_i \rightarrow \mathcal{P}$ *must satisfy the following conditions.*

1. F_i *must be affine.*

2. $\pi_i(F_i(q)) = q$ *for all* $q \in \mathcal{S}_i$.

The first condition is required by our use of integer linear programming techniques. We will allow symbolic constants in the affine part of the embedding functions. The second condition states that if point $q \in \mathcal{S}_i$ is mapped to a point $p \in \mathcal{P}$, then the component in p corresponding to $\mathcal{S}_i$ is q itself. Therefore, for our running example, we will permit embedding functions like F_1 but not F_2:

$$F_1(\begin{bmatrix} t \\ j_1 \end{bmatrix}) = \begin{bmatrix} t \\ j_1 \\ t+1 \\ t+j_1 \end{bmatrix} \quad F_2(\begin{bmatrix} t \\ j_2 \end{bmatrix}) = \begin{bmatrix} t \\ t+j_2 \\ t+1 \\ j_2 \end{bmatrix}.$$

Each F_i is therefore one-to-one, but points from two different statement iteration spaces may be mapped to a single point in the product space. Affine embedding functions can be decomposed into their linear and offset parts as follows: $F_j(i_j) = G_j i_j + g_j$.

Code Generation

Given an execution order $(\mathcal{P}, \mathcal{F})$ for a program, where $\mathcal{P}$ is the product space and $\mathcal{F}$ is a set of embedding functions satisfying Definition 3, code for executing the program in this new order can be generated as follows. We traverse the entire product space lexicographically, and at each point of $\mathcal{P}$ we execute the original program with all statements protected by guards. These guards ensure that only statement instances mapped to the current point (by the embedding functions F_i) are executed.

For example, consider the following embeddings for our running example –

$$F_1(\begin{bmatrix} t \\ j_1 \end{bmatrix}) = \begin{bmatrix} t \\ j_1 \\ t \\ j_1+1 \end{bmatrix} \quad F_2(\begin{bmatrix} t \\ j_2 \end{bmatrix}) = \begin{bmatrix} t \\ j_2-1 \\ t \\ j_2 \end{bmatrix}$$

Naive code that respects this execution order can be generated as shown in Figure 2.11(a). The outer four loops (`t1, j1', t2, j2'`) traverse the entire product space (explaining the `-inf` and `+inf` in their loop bounds). At each point (`t1, j1', t2, j2'`), the entire original program is executed with the statements protected by guards which ensure that only statement instances mapped to the current point are executed. The condition in the guard for a particular statement implements the embedding function for that statement. Of course, the code generated in this way clearly cannot be executed directly. Standard polyhedral techniques [17] must be used to find the loop-bounds and to remove redundant loops. A version of the code with the bounds determined is shown in Figure 2.11(b). This code can be further optimized by removing the conditionals in the innermost loop through index-set splitting the outer loops and eliminating redundant loops.

Embedding the Original Code

For completeness, we show that there always exists a way of embedding the code in the product space so that the original program execution order is preserved.

As an example, consider the code in Figure 2.12. It is easy to verify that this code is equivalent to our running example, and has the same execution order. Intuitively, the loops in Figure 2.12 correspond to the dimensions of the product space; the embedding functions for different statements can be read off from the guards in the loop nest:

$$F_1(\begin{bmatrix} t_1 \\ j_1 \end{bmatrix}) = \begin{bmatrix} t_1 \\ j_1 \\ t_1 \\ 2 \end{bmatrix} \quad F_2(\begin{bmatrix} t_2 \\ j_2 \end{bmatrix}) = \begin{bmatrix} t_2 \\ N-1 \\ t_2 \\ j_2 \end{bmatrix}.$$

```
for t1  = -inf, +inf
for j1' = -inf, +inf
for t2  = -inf, +inf
for j2' = -inf, +inf
    for t = 1, M
        for j1 = 2, N-1
            if (t1 == t && j1' == j1 && t2 == t && j2' == j1+1)
S1:            L(j1) = A(j1-1) + A(j1+1)
            endif
        end
        for j2 = 2, N-1
            if (t1 == t && j1' == j2-1 && t2 == t && j2' == j2)
S2:            A(j2) = L(j2)
            endif
        end
    end
end
end
end
end
```

(a) Naive Code

```
for t1 = 1, M
    for j1 = 1, N-1
        for t2 = 1, M
            for j2 = 2, N
                if (t2 == t1 && j2 == j1+1 && j1 > 1)
S1:                 L(j1) = A(j1-1) + A(j1+1)
                endif
                if (t1 == t2 && j1 == j2-1 && j2 < N)
S2:                 A(j2) = L(j2)
                endif
            end
        end
    end
end
```

(b) Code with Bounds

Figure 2.11: Code Generation from Product Space and Embeddings

To preserve the original program execution order, the embedding functions in this example are chosen to satisfy the following conditions: (i) the identity mapping

```
for t1 = 1, M
    for j1 = 2, N-1
        for t2 = 1, M
            for j2 = 2, N-1
                if (t2 == t1 && j2 == 2)
S1:                 L(j1) = A(j1-1) + A(j1+1)
                endif
                if (t1 == t2 && j1 == N-1)
S2:                 A(j2) = L(j2)
                endif
            end
        end
    end
end
```

Figure 2.12: Original Embeddings for 1D relaxation

is used for dimensions corresponding to common loops (t) in the original code; and (ii) mappings for dimensions corresponding to non-common loops are chosen to preserve the original execution order.

In general, we can embed any code into its product space as follows. Let $F_k : \mathcal{S}_k \rightarrow \mathcal{P}$ be the affine function that maps the statement $\mathtt{S}_k$ to the product space. The components of F_k that map into the dimensions of the product space corresponding to statement $\mathtt{S}_j$ are denoted by $F_{k,j}$. Our initial requirement on embedding functions can be summarized by $F_{k,k}(\vec{\imath}_k) = \vec{\imath}_k$. We will use $\vec{\imath}_k$ to represent the vector of loop index values surrounding the point i_k.

We define $common(k,l)$ to be a function that returns the loop index variables of the loops common to both $\vec{\imath}_k$ and $\vec{\imath}_l$. Similarly, we define $noncommon(k,l)$ to return the loop index variables of the rest of the loops in $\vec{\imath}_l$. For the example above, we have $common(1,2) = (t_1)^T$, $common(2,1) = (t_2)^T$, $noncommon(1,2) = (j_2)^T$ and $noncommon(2,1) = (j_1)^T$.

Let $\min_{\prec}(\vec{\imath})$ and $\max_{\prec}(\vec{\imath})$ return the lexicographically smallest and largest values of the indices of the loops $\vec{\imath}$. For our example, $\min_{\prec}(j_2) = 2$ and $\max_{\prec}(j_1) = N-1$.

For every statement $\mathtt{S}_l$ that occurs syntactically after $\mathtt{S}_k$ in the original program (i.e. $l > k$) we define

$$F_{k,l}(\vec{\imath}_k) = \begin{bmatrix} common(k,l) \\ \min_{\prec}(noncommon(k,l)) \end{bmatrix},$$

and for every statement $\mathtt{S}_l$ that occurs syntactically before $\mathtt{S}_k$ in the original program (i.e. $l < k$) we define

$$F_{k,l}(\vec{\imath}_k) = \begin{bmatrix} common(k,l) \\ \max_{\prec}(noncommon(k,l)) \end{bmatrix}.$$

These embeddings represent the original execution order.

2.3.3 Examples of Embeddings

The pair $(\mathcal{P}, \mathcal{F})$ as restricted by Section 2.3.2 can only represent a restricted set of execution orders. Under these restrictions, the following loop transformations cannot be represented –

1. Index-Set-Splitting : Since all the instances of a particular statement are mapped with the same affine embedding function, index-set-splitting is no longer possible.

2. Tiling requires the introduction of additional dimensions and pseudo-linear embeddings, and hence cannot be represented using the product space and affine embeddings.

On the other hand, the product space and affine embeddings are sufficient to capture most common loop transformations, like code-sinking, loop-fission and loop-fusion that are used in current compilers such as the SGI MIPSPro to convert

```
for t1 = 1, M
   for j1 = 2, N-1
S1:    L(j1) = A(j1-1) + A(j1+1)
   end
end
for t2 = 1, M
   for j2 = 2, N-1
S2:    A(j2) = L(j2)
   end
end
```

(a) Fissioned code

```
for t1 = 1, M
    for j1 = 2, N-1
        for t2 = 1, M
            for j2 = 2, N-1
                if (t2 == 1 && j2 == 2)
S1:                 L(j1) = A(j1-1) + A(j1+1)
                endif
                if (t1 == M && j1 == N-1)
S2:                 A(j2) = L(j2)
                endif
            end
        end
    end
end
```

(b) Transformed code

$$F_1(\begin{bmatrix} t_1 \\ j_1 \end{bmatrix}) = \begin{bmatrix} t_1 \\ j_1 \\ 1 \\ 2 \end{bmatrix} \quad F_2(\begin{bmatrix} t_2 \\ j_2 \end{bmatrix}) = \begin{bmatrix} M \\ N-1 \\ t_2 \\ j_2 \end{bmatrix}$$

(c) Embeddings

Figure 2.13: Embeddings for Loop Fission

```
for t1 = 1, M
   for j = 2, N-1
S1:    L(j) = A(j-1) + A(j+1)
S2:    A(j) = L(j)
   end
end
```

(a) Fused code

```
for t1 = 1, M
    for j1 = 2, N-1
        for t2 = 1, M
            for j2 = 2, N-1
                if (t2 == t1 && j2 == j1)
S1:                 L(j1) = A(j1-1) + A(j1+1)
                endif
                if (t1 == t2 && j1 == j2)
S2:                 A(j2) = L(j2)
                endif
            end
        end
    end
end
```

(b) Transformed code

$$F_1(\begin{bmatrix} t_1 \\ j_1 \end{bmatrix}) = \begin{bmatrix} t_1 \\ j_1 \\ t_1 \\ j_1 \end{bmatrix} \quad F_2(\begin{bmatrix} t_2 \\ j_2 \end{bmatrix}) = \begin{bmatrix} t_2 \\ j_2 \\ t_2 \\ j_2 \end{bmatrix}$$

(c) Embeddings

Figure 2.14: Embeddings for Loop Fusion

```
for t1 = 1, M
    for j = 2, 2
S1:     L(j) = A(j-1) + A(j+1)
    end
    for j = 3, N-1
S1:     L(j) = A(j-1) + A(j+1)
S2:     A(j-1) = L(j-1)
    end
    for j = N-1, N-1
S2:     A(N-1) = L(N-1)
    end
end
```

(a) Fused code

```
for t1 = 1, M
    for j1 = 2, N
        for t2 = 1, M
            for j2 = 1, N-1
                if (t2 == t1 && j2 == j1-1 &&  j1 <= N-1)
S1:                 L(j1) = A(j1-1) + A(j1+1)
                endif
                if (t1 == t2 && j1 == j2+1 &&  j2 >= 2)
S2:                 A(j2) = L(j2)
                endif
            end
        end
    end
end
```

(b) Transformed code

$$F_1(\begin{bmatrix} t_1 \\ j_1 \end{bmatrix}) = \begin{bmatrix} t_1 \\ j_1 \\ t_1 \\ j_1 - 1 \end{bmatrix} \quad F_2(\begin{bmatrix} t_2 \\ j_2 \end{bmatrix}) = \begin{bmatrix} t_2 \\ j_2 + 1 \\ t_2 \\ j_2 \end{bmatrix}$$

(c) Embeddings

Figure 2.15: Embeddings for Skewed Loop Fusion

imperfectly-nested loop nests into perfectly-nested ones. Tiling can later be used to further transform the product space.

Figure 2.13 illustrates this for loop fission. After loop fission, all instances of statement `S1` in Figure 2.10 are executed before all instances of statement `S2`. The resulting code is shown in Figure 2.13(a). It is easy to verify that this effect is achieved by the transformed code of Figure 2.13(b). Intuitively, the loop nest in this code corresponds to the product space; the embedding functions for different statements can be read off from the guards in this loop nest and are shown in Figure 2.13(c). Note that this execution order is not legal for the original program.

In a similar manner, Figure 2.14 illustrates how the `j1` and `j2` loops can be fused together. The resulting code is shown in Figure 2.14(a). The transformed code in Figure 2.14(b) has the same effect. The embeddings chosen (shown in Figure 2.14) map the statement instances $S_1(t, j)$ and $S_2(t, j)$ to the same point (t, j, t, j) in the product space. These embeddings are shown in Figure 2.14(c). Note that fusing the two loops is not legal – in order to fuse them legally, we need to peel away the first iteration of the `j1` loop, the last iteration of the `j2` loop, and then fuse the remaining sections. This execution order and the corresponding embedding that achieves this is shown in Figure 2.15. Such a transformation is known as *skewed fusion*.

The transformed code corresponding to the original execution order shown in Figure 2.12 is an example of a generalized version of code-sinking.

Dimension of Product Space

The number of dimensions in the product space can be quite large, and one might wonder if it is possible to embed statement iteration spaces into a smaller space without restricting program transformations. For example, in Figure 2.14(b), statements in the body of the transformed code are executed only when `j2 = j1`, so it is possible to eliminate the `j2` loop entirely, replacing all occurrences of `j2` in the body by `j1`. Therefore, dimension j_2 of the product space is redundant, as is t_2. More generally, we can state the following result.

Theorem 1 *Let $\mathcal{P}'$ be any space and let $\{F_1, F_2, \ldots, F_n\}$ be a set of affine embedding functions $F_j : \mathcal{S}_j \to \mathcal{P}'$ satisfying the conditions in Definition 3.*
Let $F_j(i_j) = G_j i_j + g_j$. The number of independent dimensions of the space $\mathcal{P}'$ is equal to the rank of the matrix $G = [G_1 G_2 \ldots G_n]$.

In Figure 2.14, the rank of this matrix

$$G = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

is 2, which is also the number of independent dimensions in the product space. The remaining 2 dimensions are redundant.

Corollary 1 *Let $\mathcal{P}$ be the product space.*

1. *Any space $\mathcal{P}'$ bigger than $\mathcal{P}$ has redundant dimensions under any set of affine embedding functions.*

2. *There exist affine embedding functions $\{F_1, F_2, \ldots, F_n\}$ for which no dimension of $\mathcal{P}$ is redundant.*

Intuitively, Corollary 1 states that the product space is "big enough" to model any affine transformation of the original code. Furthermore, there are affine transformations that utilize all dimensions of the product space. For example, there are no redundant dimensions in the product space of completely fissioned code, as Figure 2.13 illustrates. The corresponding matrix G for this code is shown below :

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In general, therefore, it is the embeddings that determine whether there are redundant dimensions in the product space.

2.4 Transformed Product Spaces and Valid Embeddings

Our definition of the product space ($\mathcal{P}$) has fixed the order of the dimensions. Clearly this might not be the best order that reduces reuse distances. In order to circumvent this restriction, we will need to also consider other program execution orders obtained by traversing the product space in ways other than lexicographic traversal.

If p is the dimension of the product space, let $T^{p\times p}$ be a unimodular matrix. Any such matrix defines an order in which the points of the product space are visited.

Definition 4 *The space resulting from transforming the product space $\mathcal{P}$ by a unimodular matrix T is called the* transformed product space *under transformation T.*

For a set of embedding functions $\mathcal{F} = \{F_1, F_2, \ldots F_n\}$ and a transformation matrix T, we model execution of the transformed code by walking the transformed product space lexicographically and executing all statement instances mapped to each point as we visit it. For this to be *legal*, a lexicographic order of traversal of the transformed product space must satisfy all dependencies. To formulate this condition, it is convenient to define the following concept.

Definition 5 *Let $\{F_1, F_2, \ldots F_n\}$ be a set of embedding functions for a program, and let $T^{p \times p}$ be a unimodular matrix. Let*

$$\mathcal{D} : D \begin{bmatrix} i_s \\ i_d \end{bmatrix} + d \geq 0$$

be a dependence class for this program. The difference vector *for a pair $(i_s, i_d) \in \mathcal{D}$ is the vector*

$$V_{\mathcal{D}}(i_s, i_d) \equiv [F_d(i_d) - F_s(i_s)] \, .$$

The set of difference vectors for all points in a dependence class $\mathcal{D}$ will be called the difference vectors for $\mathcal{D}$*; abusing notation, we will refer to this set as $V_{\mathcal{D}}$.*

The set of all difference vectors for all dependence classes of a program will be called the difference vectors of that program*; we will refer to this set as V.*

With these definitions, it is easy to express the condition under which a lexicographic order of traversal of the transformed product space respects all program dependences.

Definition 6 *Let $T^{p \times p}$ be a unimodular matrix. A set of embedding functions $\{F_1, F_2, \ldots, F_n\}$ is said to be* valid *for T if $Tv \succeq 0$ for all $v \in V$.*

2.4.1 Determining Valid Embeddings

In this section, we show how to determine valid embedding functions for a given unimodular matrix T. In the next chapter, we will show how to determine T.

As defined previously, a set of embedding functions $\mathcal{F} = \{F_1, F_2, \dots, F_n\}$ is said to be *valid* for a traversal order T if for every difference vector v, the vector Tv is lexicographically positive.

We shall determine the embedding functions dimension by dimension.

For a given embedding $\mathcal{F}$, a dimension is said to *satisfy* all the difference vectors $V_{\mathcal{D}}$ of a class $\mathcal{D}$ if the corresponding entry of the vector Tv is strictly positive and there exists atleast one difference vector v in $V_{\mathcal{D}}$ for which the entries of Tv corresponding to the previous dimensions are all zero. Intuitively, once the embeddings to this dimension have been determined, all the difference vectors corresponding to the dependence class $\mathcal{D}$ will be lexicographically positive. The dependence class $\mathcal{D}$ is said to be *satisfied*.

In determining the embedding function for the j^{th} dimension $\mathcal{F}^j$, we do not need to consider any of the satisfied dependence classes. For every dependence pair (i_s, i_d) in every unsatisfied dependence class $\mathcal{D} : D \begin{bmatrix} i_s \\ i_d \end{bmatrix} + d \geq 0$, we require that

$$T(\mathcal{F}_d^j(i_d) - \mathcal{F}_s^j(i_s)) \geq 0$$

For affine embedding functions, the above condition can be written as follows:

$$T \begin{bmatrix} -G_s^j & G_d^j \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} + T \left[g_d^j - g_s^j \right] \geq 0.$$

We use Farkas' lemma in order to obtain the set of valid embedding functions for the j^{th} dimension.

Lemma 1 *(Farkas) Any affine function $f(x)$ which is non-negative everywhere over a polyhedron defined by the inequalities $Ax + b \geq 0$ can be represented as follows:*

$$f(x) = \lambda_0 + \Lambda^T Ax + \Lambda^T b$$
$$\lambda_0 \geq 0, \Lambda \geq 0$$

where Λ is a vector of length equal to the number of rows of A. λ_0 and Λ are called the Farkas multipliers.

Applying Farkas' Lemma to our dependence equations we obtain

$$\begin{aligned} T \begin{bmatrix} -G_s^j & G_d^j \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} + T \left[g_d^j - g_s^j \right] & \\ = \; & y + Y^T D \begin{bmatrix} i_s \\ i_d \end{bmatrix} + Y^T d \\ & y \geq 0, Y \geq 0, \end{aligned}$$

where the vector y and the matrix Y are the Farkas multipliers.

Equating coefficients of i_s, i_d on both sides, we get

$$\begin{aligned} T \begin{bmatrix} -G_s^j & G_d^j \end{bmatrix} &= Y^T D \\ T \left[g_d - g_s \right] &= y + Y^T d \\ y \geq 0, Y &\geq 0. \end{aligned} \tag{2.2}$$

The Farkas multipliers in System (2.2) can be eliminated through Fourier-Motzkin projection to give a system of inequalities constraining the unknown embedding coefficients. This defines the set of possible embedding functions $\mathcal{F}^j$ given embedding functions to the previous $j - 1$ dimensions.

We will illustrate the above with the triangular solve example in Figure 2.6. The embedding functions for this example are as shown below –

$$F_1(\begin{bmatrix} c_1 \\ r_1 \\ k_1 \end{bmatrix}) = \begin{bmatrix} c_1 \\ r_1 \\ k_1 \\ f_1^{c_2} \\ f_1^{r_2} \end{bmatrix} \quad F_2(\begin{bmatrix} c_2 \\ r_2 \end{bmatrix}) = \begin{bmatrix} f_2^{c_1} \\ f_2^{r_1} \\ f_2^{k_1} \\ c_2 \\ r_2 \end{bmatrix}$$

where $f_1^{c_2}$ etc. are unknown affine functions that must be determined. Assume that T is the identity matrix.

As discussed in Section 2.1.2, the code has two dependence classes –

$$\begin{array}{lrcccccccccc} \mathcal{D}_1: & \mathtt{M} & \geq & c_1 & \geq & 1 & \mathtt{M} & \geq & c_2 & \geq & 1 \\ & \mathtt{N} & \geq & r_1 & \geq & 1 & \mathtt{N} & \geq & r_2 & \geq & 1 \\ & r_1 - 1 & \geq & k_1 & \geq & 1 \\ & r_1 & = & r_2 \\ & c_1 & = & c_2 \end{array}$$

$$\begin{array}{lrcccccccccc} \mathcal{D}_2: & \mathtt{M} & \geq & c_1 & \geq & 1 & \mathtt{M} & \geq & c_2 & \geq & 1 \\ & \mathtt{N} & \geq & r_1 & \geq & 1 & \mathtt{N} & \geq & r_2 & \geq & 1 \\ & r_1 - 1 & \geq & k_1 & \geq & 1 \\ & k_1 & = & r_2 \\ & c_1 & = & c_2 \end{array}$$

Consider the first dimension. None of the dependence classes are satisfied yet, and hence both of them must be considered. We, therefore, have to ensure two conditions:

1. $f_2^{c_1}(c_2, r_2) - c_1 \geq 0$ for all points in $\mathcal{D}_1$, and

2. $c_1 - f_2^{c_1}(c_2, r_2) \geq 0$ for all points in $\mathcal{D}_2$.

Consider the first condition. Let $f_2^{c_1}(c_2, r_2) = g_{c_2}c_2 + g_{r_2}r_2 + g_M M + g_N N + g_1$. Applying Farkas' Lemma, we get $f_2^{c_1}(c_2, r_2) - c_1 = \lambda_0 + \lambda_1(M - c_1) + \lambda_2(c_1 - 1) + \cdots + \lambda_{13}(c_1 - c_2) + \lambda_{14}(c_2 - c_1)$ where $\lambda_0, \ldots, \lambda_{14}$ are non-negative[5]. Projecting the λ's out, we find out that the coefficients of $f_2^{c_1}(c_2, r_2)$ must satisfy the following inequalities:

$$\begin{aligned} g_M &\geq 0 \\ g_N &\geq 0 \\ g_{c_2} + g_M &\geq 1 \\ g_{r_2} + g_N &\geq 0 \\ g_{c_2} + 2g_{r_2} + g_M + 2g_N + g_1 &\geq 1 \end{aligned}$$

Similarly, for the second condition, this procedure determines the following constraints:

$$\begin{aligned} g_M &\leq 0 \\ g_N &\leq 0 \\ g_{c_2} + g_M &\leq 1 \\ g_{r_2} + g_N &\leq 0 \\ g_{c_2} + g_{r_2} + g_M + 2g_N + g_1 &\leq 1 \end{aligned}$$

The conjunction of these inequalities gives the solution $f_2^{c_1}(c_2, r_2) = c_2$.

This choice of embedding function for the first dimension does not satisfy either of the two dependence classes – the first dimension of of all difference vectors belonging

[5]There are 14 inequalities that define $\mathcal{D}_1$ in , so there are 14 Farkas multipliers $\lambda_1 \ldots \lambda_{14}$.

to these two classes is 0. Hence, both these classes must again be considered in determining the set of valid embedding function for the second dimension.

```
T   := Transformation matrix

ALGORITHM DetermineValidEmbeddings (T)
DU  := Set of unsatisfied dependence classes
             (initialized to all dependence classes of program)
DS  := Set of satisfied dependence classes for the current layer
             (initialized to empty set)

for dimension j = 1,p of the product space

      Construct system S constraining the jth dimension
         of every embedding function as follows:
         for each unsatisfied dependence class u ∈ DU
            Add constraints so that each entry in dimension j of
            all transformed difference vectors of u is  non-negative;
         endfor

      if system has solutions
         Pick a solution
         Update DS and DU;
         Continue j loop;
      endif

      // if the previous system does not have a solution
      goto no_solution
endfor

return embeddings

no_solution :
      Error. No solution found.
```

Figure 2.16: Algorithm to Determine Valid Embeddings

The algorithm to determine valid embedding functions is shown in Figure 2.16. The main loop of the algorithm proceeds dimension by dimension constraining the unknown coefficients of the embedding functions so that no difference vector in an unsatisfied dependence class is violated. If the resulting linear system has solutions, then one of the possible solutions is picked non-deterministically. Otherwise, no solution is possible for the given transformation matrix T and the embeddings chosen for the previous dimensions.

Chapter 3

Using the Framework

3.1 Embeddings that permit Tiling

The framework described in the previous chapter can be used to find embeddings that allow the tiling of imperfectly-nested loop nests. The intuitive idea is to embed all statement iteration spaces in the product space, and then tile the product space after transforming it if necessary by a unimodular transformation. Tiling is legal if the transformed product space is *fully permutable*—that is, if its dimensions can be permuted arbitrarily without violating dependences. This approach is a generalization of the approach used to tile perfectly-nested loop nests [32, 23]; the embedding step is not required for perfectly-nested loop nests because all statements have the same iteration space to begin with.

3.1.1 Determining Constraints on Embeddings and Transformations

The condition for full permutability of the transformed product space is the following.

Lemma 2 *Let $\{F_1, F_2, \ldots, F_n\}$ be a set of embeddings, and let T be a unimodular matrix. The transformed product space is fully permutable if $v \geq 0$ for all $v \in V$.*

The proof of this result is trivial: if every entry in every difference vector is non-negative, the space is fully permutable, so it can be tiled. Thus our goal is to find embeddings F_i and a product space transformation T that satisfy the condition of Lemma 2.

Let $\mathcal{D} : D \begin{bmatrix} i_s \\ i_d \end{bmatrix} + d \geq 0$ be any dependence class. For affine embedding functions, the condition $v \geq 0$ in Lemma 2 can be written as follows:

$$T \begin{bmatrix} -G_s & G_d \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} + T\,[g_d - g_s] \geq 0.$$

The affine form of Farkas' Lemma lets us express the unknown matrices T,G_s,g_s,G_d and g_d in terms of D.

Applying Farkas' Lemma to our dependence equations we obtain

$$\begin{aligned} T \begin{bmatrix} -G_s & G_d \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} \quad + \quad & T\,[g_d - g_s] \\ = \quad & y + Y^T D \begin{bmatrix} i_s \\ i_d \end{bmatrix} + Y^T d \\ y \geq 0, Y \geq 0, & \end{aligned}$$

where the vector y and the matrix Y are the Farkas multipliers.

Equating coefficients of i_s, i_d on both sides, we get

$$\begin{aligned} T\begin{bmatrix} -G_s & G_d \end{bmatrix} &= Y^T D \\ T\,[g_d - g_s] &= y + Y^T d \\ y \geq 0, Y &\geq 0. \end{aligned} \tag{3.1}$$

The Farkas multipliers in System (3.1) can be eliminated through Fourier-Motzkin projection to give a system of inequalities constraining the unknown embedding coefficients and transformation matrix. Since we require that all difference vector elements be non-negative, we can apply this procedure to each dimension of the product space separately.

Applying the above procedure to all dependence classes results in a system of inequalities constraining the embedding functions and transformation. A fully permutable product space is possible if and only if that system has a solution. The set of dimensions for which the equations have a solution will constitute a fully permutable sub-space of the product space.

3.1.2 Solving for Embeddings and Transformations

In System (3.1), T is unknown while each G_i is partially specified[1]. To solve such systems, we will heuristically restrict T and solve the resulting linear system for appropriate embeddings if they exist.

We will initially restrict T to be the identity matrix. In general, it may not be possible to find embeddings that make the product space fully permutable (that is, with T restricted to the identity matrix). For such programs, transforming the

[1] The embedding functions are partially fixed because of condition (2) in Definition 3.

product space by a non-trivial transformation T may result in a fully permutable space that can be tiled. This is the case for the relaxation codes discussed in Section 3.5. If we fail to find embeddings with T restricted to the identity matrix, we can try to find combinations of loop permutation, reversal and skewing for which it can find valid embeddings.

Loop reversal for a given dimension of the product space is handled by requiring the entry in that dimension of each difference vector to be non-positive. For a dependence class $\mathcal{D}$, the condition that the j^{th} entry of all of its difference vectors $V_{\mathcal{D}}$ are non-positive can be written as follows:

$$\begin{bmatrix} -G_s^j & G_d^j \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} + g_d^j - g_s^j \leq 0 \tag{3.2}$$

which is equivalent to

$$\begin{bmatrix} G_s^j & -G_d^j \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} + g_s^j - g_d^j \geq 0. \tag{3.3}$$

Loop skewing is handled as follows. We replace the non-negativity constraints on the j^{th} entries of all difference vectors in V by linear constraints that guarantee that these entries are bounded below by a negative constant, as follows:

$$\begin{bmatrix} -G_s^j & G_d^j \end{bmatrix} \begin{bmatrix} i_s \\ i_d \end{bmatrix} + g_d^j - g_s^j + \alpha \geq 0, \quad \alpha \geq 0 \tag{3.4}$$

where α is an additional variable introduced into the system. The smallest value of α that satisfies this system can be found by projecting out the other variables and picking the lower bound of α. If the system has a solution, the negative entries in the j^{th} entry of all difference vectors are bounded by the value of α. If every difference vector that has a negative value in dimension j, has a strictly positive entry in a dimension preceding j, loop skewing can be used to make all entries in dimension j positive.

3.2 Embeddings that Enhance Reuse

Consider a reuse class $\mathcal{R}$ and a reuse pair $(i_s, i_d) \in \mathcal{R}$. We will make it explicit that the points i_s and i_d can also be represented by vectors by refering to them as $\vec{\imath}_s$ and $\vec{\imath}_d$ respectively. The locality enhancement model in Section 2.2 required the minimization of $Distance(\vec{\imath}_s, \vec{\imath}_d)$, which is the number of points in the space $\mathcal{P}$ with statements mapped to them between $F_s(\vec{\imath}_s)$ and $F_d(\vec{\imath}_d)$. Unfortunately, it is not possible to calculate $Distance(\vec{\imath}_s, \vec{\imath}_d)$ efficiently, since there may be points with no statements mapped to them. Instead, we reduce reuse distances as follows.

Consider the *reuse vector* ($\vec{v}$) for the reuse pair $(\vec{\imath}_s, \vec{\imath}_d)$ for a given choice of embedding functions $\mathcal{F} = \{F_1, F_2, \dots, F_n\}$; we will refer to the j^{th} entry of this vector as v_j.

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_p \end{bmatrix} = F_d(\vec{\imath}_d) - F_s(\vec{\imath}_s) = \begin{bmatrix} F_{d,1}(\vec{\imath}_d) - F_{s,1}(\vec{\imath}_s) \\ F_{d,2}(\vec{\imath}_d) - F_{s,2}(\vec{\imath}_s) \\ \vdots \\ F_{d,s}(\vec{\imath}_d) - \vec{\imath}_s \\ \vdots \\ \vec{\imath}_d - F_{s,d}(\vec{\imath}_s) \\ \vdots \\ F_{d,n}(\vec{\imath}_d) - F_{s,n}(\vec{\imath}_s) \end{bmatrix}.$$

We say that dimension j *carries reuse* for the reuse pair $(\vec{\imath}_s, \vec{\imath}_d)$ if $v_j \neq 0$. If a dimension carries reuse for some reuse pair in a reuse class $\mathcal{R}$, that dimension is said to carry reuse for that reuse class.

For all reuse pairs $(\vec{\imath}_s, \vec{\imath}_d) \in \mathcal{R}$, entries corresponding to $F_{d,k}(\vec{\imath}_d) - F_{s,k}(\vec{\imath}_s)$ (for $k \neq s, d$) can be made zero simultaneously (e.g. by choosing $F_{d,k}(\vec{\imath}_d) = F_{s,k}(\vec{\imath}_s) = const$). This may not always be possible for the elements $F_{d,s}(\vec{\imath}_d) - \vec{\imath}_s$ and $\vec{\imath}_d - F_{s,d}(\vec{\imath}_s)$ since the appropriate functions $F_{d,s}$ and $F_{s,d}$ may not exist. We try to make these entries zero; if this does not succeed, we can permute these dimensions of the product space so that they are innermost and tile them. This results in the following strategy:

1. We attempt to make all entries v_j of the reuse vector zero by choosing embedding functions appropriately. Since the dimensions of the embedding functions are independent, we can process each dimension separately. If we succeed in making all entries $v_j = 0$, then the reuse distance is also zero.

2. We reorder the dimensions of the product space so that dimensions for which $v_j = 0$ come first, and dimensions with larger entries come later.

3. We reduce reuse distances further by *tiling* all dimensions j for which the entry v_j of the reuse vector is non-zero.

3.3 Algorithm

```
for i = 1, N
    for j = 1, N
S1:     c(i,j) = 0
        for k = 1, N
S2:         c(i,j) = c(i,j) + a(i,k) * b(k,j)
        end
    end
end
```

Figure 3.1: Imperfectly-nested MMM

Before presenting the general locality enhancement algorithm, we illustrate our approach on the program of Figure 3.1. The iteration space $\mathcal{S}_1$ of statement `S1` is a two-dimensional space $i_1 \times j_1$, while the iteration space $\mathcal{S}_2$ of `S2` is a three-dimensional space $i_2 \times j_2 \times k_2$. The product space is the five dimensional space $i_1 \times j_1 \times i_2 \times j_2 \times k_2$.

There are two dependence classes in this example:

1. Dependence class $\mathcal{D}_1 = \{(i_1, j_1, i_2, j_2, k_2) : 1 \leq i_1, j_1, i_2, j_2, k_2 \leq N, i_1 = i_2, j_1 = j_2\}$ is a flow-dependence that arises because statement `S1` writes to a location `c(i,j)` which is then read by statement `S2`.

2. Dependence class $\mathcal{D}_2 = \{(i_2, j_2, k_2, i_2', j_2', k_2') : 1 \leq i_2, j_2, k_2, i_2', j_2', k_2' \leq N, i_2 = i_2', j_2 = j_2', k_2 < k_2'\}$ is a flow-dependence that arises because statement `S2` writes to location `c(i,j)` which is then read by this statement in a later k iteration. This dependence also captures the anti- and output-dependences of statement `S2` on itself.

These two classes also represent reuse classes. The program has other reuse classes arising from spatial locality and input dependences, but these are not shown here for simplicity.

Our locality enhancement algorithm will

1. determine affine embedding functions,

2. transform the product space,

3. eliminate redundant dimensions, and

4. decide which dimensions to tile.

The most difficult steps are (1) and (2), and these are interleaved in the algorithm described in Section 3.3.4. To simplify the presentation, let us assume for now that an oracle determines the transformation of the product space in Step (2) (we show in Section 3.3.4 that interleaving eliminates the need for such an oracle). Therefore, we are left with the problem of determining affine embedding functions. These are determined one dimension at a time by solving a system of linear constraints on the coefficients of the embedding functions for that dimension. These linear constraints describe the requirements that embeddings should (i) result in a legal program, and (ii) minimize reuse distances.

For the running example, we will assume that the oracle tells us that the transformation is the identity transformation, so the product space is left unchanged. Since the product space for this program is the five dimensional space $i_1 \times j_1 \times i_2 \times j_2 \times k_2$, Definition 3 of embedding functions requires that the embedding functions for this program look like the following:

$$F_1(\begin{bmatrix} i_1 \\ j_1 \end{bmatrix}) = \begin{bmatrix} i_1 \\ j_1 \\ G^3_{i_1} i_1 + G^3_{j_1} j_1 + g^3_N N + g^3_1 \\ G^4_{i_1} i_1 + G^4_{j_1} j_1 + g^4_N N + g^4_1 \\ G^5_{i_1} i_1 + G^5_{j_1} j_1 + g^5_N N + g^5_1 \end{bmatrix}$$

$$F_2(\begin{bmatrix} i_2 \\ j_2 \\ k_2 \end{bmatrix}) = \begin{bmatrix} G^1_{i_2} i_2 + G^1_{j_2} j_2 + G^1_{k_2} k_2 + g^1_N N + g^1_1 \\ G^2_{i_2} i_2 + G^2_{j_2} j_2 + G^2_{k_2} k_2 + g^2_N N + g^2_1 \\ i_2 \\ j_2 \\ k_2 \end{bmatrix}$$

The unknowns G and g will be referred to as the *unknown embedding coefficients.*

3.3.1 First Dimension

We first find embedding coefficients for the first dimension i_1.

Legality

At the very least, these embeddings must not violate legality. Therefore, as discussed in Section 2.4, the embedding coefficients must satisfy the following constraints.

1. $G^1_{i_2} i_2 + G^1_{j_2} j_2 + G^1_{k_2} k_2 + g^1_N N + g^1_1 - i_1 \geq 0$ for all points in $\mathcal{D}_1$, and

2. $G^1_{i_2} i'_2 + G^1_{j_2} j'_2 + G^1_{k_2} k'_2 + g^1_N N + g^1_1 - (G^1_{i_2} i_2 + G^1_{j_2} j_2 + G^1_{k_2} k_2 + g^1_N N + g^1_1) \geq 0$ for points in $\mathcal{D}_2$.

Standard integer linear programming techniques can be used to convert these constraints into the following system of linear inequalities on the unknown embedding coefficients, by applying Farkas' lemma as described in the previous chapter.

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} G^1_{i_2} \\ G^1_{j_2} \\ G^1_{k_2} \\ g^1_N \\ g^1_1 \end{bmatrix} \geq \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \tag{3.5}$$

Minimizing Reuse Distance

System (3.5) clearly has many solutions. We need to choose the solution that maximizes reuse. For our running example, consider locality optimization for the reuse that arises because of dependence $\mathcal{D}_1$. To ensure that dimension i_1 does not carry reuse for $\mathcal{D}_1$, we require that

$$G^1_{i_2} i_2 + G^1_{j_2} j_2 + G^1_{k_2} k_2 + g^1_N N + g^1_1 - i_1 = 0$$

for all points $(i_1, j_1, i_2, j_2, k_2) \in \mathcal{D}_1$. This condition too can obviously be converted into a system of inequalities on the unknown coefficients of the embeddings. The conjunction of this system and System (3.5) results in the following solution:

$$G^1_{i_2} = 1, G^1_{j_2} = 0, G^1_{k_2} = 0, g^1_N = 0, g^1_1 = 0$$

Therefore, the first dimensions of the two embedding functions are $F^1_1(i_1, j_1) = i_1$ and $F^1_2(i_2, j_2, k_2) = i_2$. Intuitively, this solution fuses dimensions i_1 and i_2 of the product space.

Even in our simple example, there are other reuse classes such as $\mathcal{D}_2$. To optimize locality for more than one reuse class, we prioritize the reuse classes heuristically and try to find embedding functions that make entries of the reuse vectors of the highest-priority reuse class equal to zero. Reuse classes are considered in order of priority until all embedding coefficients for that dimension are completely determined. If we assume that reuse class $\mathcal{D}_1$ has highest priority, we see that it completely determines the first dimension of the embedding functions, so no other reuse classes can be considered.

3.3.2 Remaining Dimensions

The remaining dimensions of the embedding functions are determined successively in a manner similar to the first one. The only difference is that some of the dependence classes may already be satisfied by preceding dimensions; these do not have to be considered for legality but only for reducing reuse distances by tiling.

Let us assume that the first $j-1$ dimensions of the embedding functions $\mathcal{F}^{1:j-1}$ have been determined and that we are currently processing the j^{th} dimension of the product space.

Legality

Generalizing the corresponding notion in perfectly-nested loops, we say that a dependence class $\mathcal{D} : D \begin{bmatrix} i_s \\ i_d \end{bmatrix} + d \geq 0$ is *satisfied* by the first $j-1$ dimensions of the embedding functions $\mathcal{F}^{1:j-1}$ if the difference vector $F_d^{1:j-1}(\vec{\imath}_d) - F_s^{1:j-1}(\vec{\imath}_s)$ is lexicographically positive for all $(\vec{\imath}_s, \vec{\imath}_d) \in \mathcal{D}$. This means that this dependence will be respected regardless of how the remaining dimensions of the embedding functions are chosen. Therefore it is sufficient to require that for every pair $(\vec{\imath}_s, \vec{\imath}_d)$ in an *unsatisfied* dependence class $\mathcal{D}$,

$$F_d^j(\vec{\imath}_d) - F_s^j(\vec{\imath}_s) = \begin{bmatrix} -G_s^j & G_d^j \end{bmatrix} \begin{bmatrix} \vec{\imath}_s \\ \vec{\imath}_d \end{bmatrix} + g_d^j - g_s^j \geq 0 \tag{3.6}$$

In our running example, it can be shown that none of the dependence classes $\mathcal{D}_1$, $\mathcal{D}_2$ are satisfied by the first dimension of the embedding functions determined above, so both dependence classes must be considered when processing the second dimension.

Minimizing Reuse Distance

Constraining embedding coefficients to minimize reuse distances can be done in an identical manner to the first dimension.

An additional concern in picking coefficients for a dimension other than the first is that we may want to tile that dimension with outer dimensions. Tiling requires that these dimensions be fully permutable. We can ensure this by requiring that the constraint (3.6) holds even for satisfied dependence classes. If the resulting system has solutions, we can pick one that minimizes reuse distances as discussed for the first dimension (note that minimizing reuse distances before we add the tiling constraints might produce embeddings that do not allow tiling). If the resulting system has no solutions, the current dimension cannot be made permutable with outer dimensions, so constraint (3.6) is dropped for satisfied dependence classes.

Our algorithm produces the following embeddings for the running example:

$$F_1\left(\begin{bmatrix} i_1 \\ j_1 \end{bmatrix}\right) = \begin{bmatrix} i_1 \\ j_1 \\ i_1 \\ j_1 \\ 0 \end{bmatrix} \quad F_2\left(\begin{bmatrix} i_2 \\ j_2 \\ k_2 \end{bmatrix}\right) = \begin{bmatrix} i_2 \\ j_2 \\ i_2 \\ j_2 \\ k_2 \end{bmatrix}$$

This embedding allows all five dimensions to be tiled. The code generation algorithm determines that i_2 and j_2 are redundant, and tiles the remaining dimensions.

3.3.3 Putting it All Together

If the transformation on the product space is given, we can obtain embedding coefficients for each dimension successively by constraining them based on (i) legality, (ii) tiling considerations, and (iii) minimizing reuse distances.

The algorithm for formulating legality and tiling constraints for a given dimension q is shown in Figure 3.2. This algorithm takes the dimension being processed, and the sets of unsatisfied and satisfied dependence classes as input, and returns a linear system L expressing constraints on embedding coefficients. Figure 3.3 shows how such a linear system is further constrained to determine coefficients for good locality. This algorithm assumes that reuse classes have been sorted in decreasing order of priority using some heuristic. In our implementation, we rank reuse classes by estimating the number of reuse pairs in each class.

```
ALGORITHM LegalityConstraints(q, DU, DS )
 /*
   q is dimension being processed.
   DU is set of unsatisfied dependence classes.
   DS is set of satisfied dependence classes.
 */

  Construct system Temp constraining the qth dimension
    of every embedding function as follows:

  for each unsatisfied dependence class u ∈ DU
      Add constraints so that each entry in dimension q
      of all difference vectors of u is non-negative;
  endfor
  for each satisfied dependence class s ∈ DS
      Add constraints so that each entry in dimension q
      of all difference vectors of s + positive α
      is non-negative;
  endfor
  Use Farkas' lemma to convert system Temp into
    a system L constraining unknown embedding
    coefficients;

  Return L;
```

Figure 3.2: Formulating Linear System for Legality

3.3.4 Algorithm

Figure 3.4 shows the complete locality enhancement algorithm.

Our algorithm interleaves the determination of the transformation for the product space with the determination of embedding coefficients for each dimension, and finds bands of fully permutable dimensions.

Each iteration of the inner `for each` q-loop tries to find a dimension q of the product space which can be permuted into position j of the transformed product space. The legality of this permutation is determined by a call to procedure `LegalityConstraints` in Figure 3.2 which attempts to find legal embeddings for dimension j which permit this dimension to be permuted with dimensions in the

```
ALGORITHM PromoteReuse(q,L,RS)
 /*
   q is dimension being processed.
   L is a system constraining unknown embedding coefficients.
   RS is set of prioritized reuse classes.
 */

pL:= L
for every reuse class R in RS in priority order
    Z := System constraining unknown embedding function
         coefficients so qth dimension entries of
         all reuse vectors of class R is zero

    if (pL ∩ Z ≠ ∅)
         pL := pL ∩ Z
    endif
endfor

return any set of coefficients satisfying pL;
```

Figure 3.3: Formulating Linear Systems for Promoting Reuse

same band. If this procedure succeeds, we call procedure `PromoteReuse` in Figure 3.3 to choose embeddings with good locality. We drop out of the `for` loop when no more dimensions of the product space can be added to the current fully permutable band. All satisfied dependences are then dropped from further consideration, and a new fully permutable band is started. The algorithm terminates when all dimensions of the product space have been mapped into the transformed space.

Reordering of Dimensions

When constructing bands, the algorithm does not try to optimize the order of dimensions within a band since it adds dimensions to bands in arbitrary order. Since arbitrary order may not be best for locality, we need to reorder dimensions after all embedding coefficients have been determined. This is similar to the problem of choosing a good order for loops in a fully permutable loop nest, and any of the

```
ALGORITHM LocalityEnhancement

Q  := Set of dimensions of product space;
DU := Set of unsatisfied dependence classes
             (initialized to all dependence classes);
DS := Set of satisfied dependence classes
             (initialized to empty set);
RS := Set of reuse classes of the program
             (sorted by priority);
j  := Current dimension in transformed product space
             (initialized to 1);

while (Q is non-empty)
   for each q in Q
       L = LegalityConstraints(q, DU, DS);
       if system L has solutions
             Embedding coefficients for dimension j =
               PromoteReuse(q,L,RS);
             Update DS and DU;
             Delete q from Q;
             j = j + 1;
       endif
    endfor

   // No more dimensions q can be added to current band.
   // Start a new band of fully permutable loops.

   DS := empty set;
endwhile

Apply Algorithm  DimensionOrdering to the dimensions;
Eliminate redundant dimensions;
Tile permutable dimensions with non-zero ReusePenalty;
```

Figure 3.4: Algorithm to Enhance Locality

techniques in the literature can be used. Here we present a simple heuristic similar to *memory order* [18]. We reorder dimensions of the product space so that the dimensions with most unsatisfied reuses come last. For each dimension j of the product space, we define the *reuse penalty* of that dimension with respect to embedding functions $\{F_1^j, F_2^j, \ldots, F_n^j\}$ to be the number of reuse pairs in the classes for

```
ALGORITHM DimensionOrdering

RPO  =  i1, i2, ... ip  // ReusePenalty order
NRPO =  ∅                // nearby permutation

m = p  // number of dimensions left to process
k = 0   // number of dimensions processed
while  RPO ≠ ∅
    for dimension j = 1,m
        l = ij ∈ RPO
        Let NRPO = {r1, r2, ... rk}
        if r1, r2, ... rk, l is legal
           NRPO = {r1, r2, ... rk, l}
            RPO  = RPO − {l}
            m = m − 1
            k = k + 1
            continue while loop
        endif
    endfor
endwhile
```

Figure 3.5: Determining Dimension Ordering

which the dimension carries reuse.

$$ReusePenalty(j, \mathcal{F}) = \sum_{\mathcal{R} \text{ unsatisfied}} \|\mathcal{R}\|$$

where $\|\mathcal{R}\|$ is the number of reuse pairs in reuse class $\mathcal{R}$. Clearly sorting dimensions in $ReusePenalty$ order is not always legal. Figure 3.5 shows an algorithm that finds a nearby legal permutation. Intuitively, algorithm `DimensionOrdering` tries to order dimensions greedily so that the dimension with the smallest $ReusePenalty$ is outermost if that is legal. Otherwise, it checks whether the dimension with next smallest $ReusePenalty$ can be placed outermost. Once it finds a dimension to place outermost, it repeats the process with the remaining dimensions. It is easy to see that the algorithm will always produce a legal ordering of the dimensions, and that it will pick the $ReusePenalty$ order if that is legal.

For the running example in Figure 3.1, our algorithm places all five dimensions of the product space in a single fully permutable band. It then picks the dimension order $j_1 \times j_2 \times k_2 \times i_1 \times i_2$.

Tiling

If a set of dimensions belonging to the same fully permutable band all carry reuse, then the reuse distance can be reduced further by tiling them. We can identify dimensions that carry reuse by testing the *ReusePenalty* associated with the dimension – all dimensions with non-zero *ReusePenalty* carry reuse for some reuse class.

As discussed in Section 3.1.2, it might be legal to tile some dimensions only after skewing by outer dimensions. In this case, the non-positive values in the dependence matrix must be distances bounded by a constant (α). The factors by which the outer dimensions need to be skewed can be determined easily by standard techniques for perfectly-nested loops [36].

In the case of our running example, all five dimensions of the product space have non-zero *ReusePenalty*. Hence our algorithm will decide to tile all of them. Note that of the five dimensions two are redundant and only the remaining three dimensions need to be tiled. (The redundant dimensions are dropped from consideration.) None of these dimensions need skewing in order to be legal.

3.4 Tile Size Determination

We determine tile sizes by estimating the *data foot-print* (the amount of data touched by a tile), and requiring that it fit into the cache in consideration. We tile the product space separately for each level of the memory hierarchy (we do not tile for a particular level only if the data touched by the tile will not fit into the corresponding cache level).

Our procedure for determining tile sizes has the following steps –

1. For each point in the product space find the data accessed by each statement instance mapped to it. Since the mapping from a statement instance to the product space is one-to-one and affine, the inverse mapping can easily be determined. This, combined with the affine access functions of a data reference, enables us to calculate the data accessed by each point in the product space.

   In our running example, a point $(x_1, x_2, x_3, x_4, x_5)$ of our transformed product space has the statement instances $S_1(x_4, x_1)$ (whenever $x_2 = x_1, x_5 = x_4, x_3 = 0$) and $S_2(x_4, x_1, x_3)$ (whenever $x_2 = x_1, x_5 = x_4$) mapped to it. Hence the data accessed by this point is $c(x_4, x_1)$ from statement S_1 and $c(x_4, x_1), a(x_4, x_3), b(x_1, x_3)$ from statement S_2.

2. Group all the data accessed by a product space point into equivalence classes as follows –

   (a) References to different arrays belong in different equivalence classes.

   (b) References are assigned to the same equivalence class if they can access the same array locations (ie. if they have the same linear parts).

   For our example, we have three equivalence classes $\{S_1 : c(x_4, x_1), S_2 : c(x_4, x_1)\}$, $\{S_2 : a(x_4, x_3)\}$, $\{S_2 : b(x_1, x_3)\}$.

3. From each reference class pick a random reference which will serve as our *representative reference*.

   In our example, our representative references are $c(x_4, x_1)$, $a(x_4, x_3)$ and $b(x_1, x_3)$.

4. Determine the data touched by each representative reference in a single tile of the transformed product space parameterized by the tile size. We shall limit

ourselves to choosing a single tile size B for every dimension of the product space. Determining the data touched by a single reference is straightforward. A generalized version of this problem has been studied in [27]. More accurate solutions can be obtained by using Erhart Polynomials [9].

For our example, each representative reference accesses B^2 elements in one tile of the transformed product space. The total data foot-print of all the references is $3 * B^2$ elements. The actual memory corresponding to this is $3 * B^2$ times the size in bytes of a single element of the array.

5. The data foot-print of all the references must be less than the cache size to avoid capacity misses. This gives us an upper bound on the tile size for each cache level. In order to generate code with fewer `MIN`'s and `MAX`'s, we ensure that the tile size at each level is a multiple of the tile size at the previous level.

The above formulation makes the following simplifications :

1. All tiles of the transformed product space have the same data foot-print. This is a conservative assumption, since it results in adding the references possible from all statements to the data accessed at a single product space point.

2. Boundary effects are ignored, which is justifiable for large arrays and loop bounds.

3. Conflict misses are ignored. Various techniques have been developed to find tile sizes that avoid some forms of conflict misses [21, 10, 13], but we do not use them in our current implementation.

3.5 Experimental Results

In this section, we present results from our implementation for five important codes—Triangular Solve with multiple left-hand sides, Cholesky factorization, Jacobi kernel, Red-Black Gauss-Seidel and the Tomcatv SPECfp95 benchmark. All experiments were run on an SGI Octane workstation based on a R12000 chip running at 300MHz with 32 KB first-level data cache and an unified second-level cache of size 2 MB (both caches are two-way set associative). We present the following performance numbers for each code:

1. Performance of code produced by the SGI MIPSPro compiler (Version 7.2.1) with the "-O3" flag turned on. At this level of optimization, the SGI compiler applies the following set of transformations to the code—it converts imperfectly-nested loop nests to *singly nested loops* (SNLs) by means of fission and fusion and then applies transformations like permutation, tiling and software pipelining inner loops [34].

2. Performance of code produced by an implementation of the techniques described in this paper, and then compiled by the SGI MIPSPro compiler with flags "-O3 -LNO:blocking=off" to disable further tiling by the SGI compiler. To study the efficacy of our tile size selection algorithm we also show the performance obtained by tiling with sizes ranging from 10 to 250.

The performance numbers presented show the benefits of synthesizing a sequence of locality-optimizing transformations instead of searching for that sequence. Even though the SGI MIPSPro compiler implements all the transformations necessary to optimize our benchmarks, it does not find the right sequence of transformations, so the performance of the resulting code suffers. For Cholesky factorization, the

```
for c = 1,M
    for r = 1,N
        for k = 1,r-1
S1:         B(r,c) = B(r,c) - L(r,k)*B(k,c)
        end
S2:     B(r,c) = B(r,c)/L(r,r)
    end
end
```

Figure 3.6: Triangular Solve : Original Code

performance of our optimized code approaches the performance of hand-written libraries. The numbers also show that our two-level tile size selection scheme chooses close-to-optimal tile sizes.

3.5.1 Performance

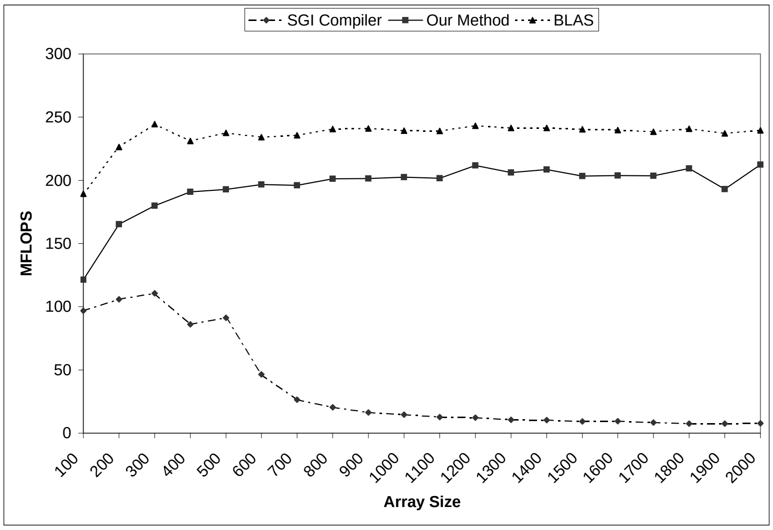
Triangular Solve

For triangular solve with multiple right-hand sides (Figure 3.6), our algorithm determines that the product space can be made fully permutable without reversal or skewing. It chooses the following embeddings after reordering and removing redundant dimensions :-

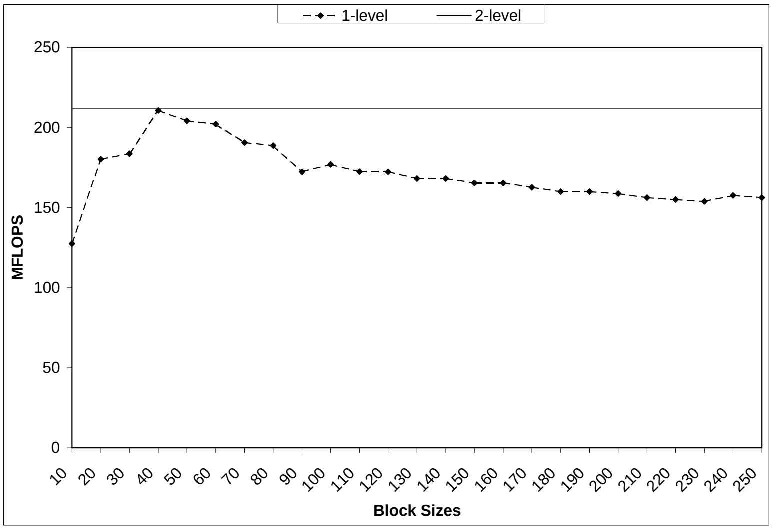
$$F_1(\begin{bmatrix} c \\ r \\ k \end{bmatrix}) = \begin{bmatrix} c \\ k \\ r \end{bmatrix} \quad F_2(\begin{bmatrix} c \\ r \end{bmatrix}) = \begin{bmatrix} c \\ r \\ r \end{bmatrix}$$

The algorithm decides that all three dimensions can be tiled. It chooses a tile size of 36 for the L1 cache and 288 for the L2 cache for all the dimensions.

Figure 3.7(a) shows performance results for a constant number of right-hand sides (`M` in Figure 3.6 is 100). The performance of code generated by our techniques is upto a factor of 10 better than the code produced by the SGI compiler, but it is still 20% slower than the hand-tuned code in the BLAS library. The high-level



(a) Performance



(b) Variation with tile size

Figure 3.7: Triangular Solve and its Performance

structure of the code we generate is similar to that of the code in the BLAS library; further improvements in the compiler-generated code must come from fine-tuning of register tiling and instruction scheduling.

Figure 3.7(b) compares the performance of our code, tiled for two levels of the memory hierarchy, with code tiled for a single level with tile sizes ranging from 10 to 250 (for a 2000×2000 array and $M = 100$). As can be seen, our two level scheme gives the best performance.

Cholesky Factorization

Cholesky factorization is used to solve symmetric positive-definite linear systems. Figure 3.8 shows one version of Cholesky factorization called *kij-Cholesky*; there are five other versions of Cholesky factorization corresponding to the permutations of the i, j, and k loops. Figure 3.9(a) compares the performance of all six versions compiled by the SGI compiler, the hand-optimized LAPACK library routine, and the code produced by our algorithm starting from *any* of the six versions.

The performance of the compiled code varies widely for the six different versions of Cholesky factorization. The *kij-Cholesky* is SNL and the SGI compiler is able to sink and tile two of the three loops (k and i), resulting in good L2 cache behavior and best performance for large matrices (about 65 MFLOPS) among the compiled codes. In contrast, the compiler is not able to optimize the *ijk-Cholesky* at all, resulting in the worst performance of about 5 MFLOPS for large matrices. The LAPACK library code performs consistently best at about 200 MFLOPS.

Our algorithm produces the same locality optimized code independent on which of the six versions we start with. That is expected as the abstraction that our algorithm uses—statements, statement iteration spaces, dependencies, and reuses—is the same

```
for k = 1,N
S1: a(k,k) = sqrt(a(k,k))
    for i = k+1,N
S2:     a(i,k) = a(i,k) / a(k,k)
        for j = k+1,i
S3:         a(i,j) -= a(i,k) * a(j,k)
        end
    end
end
```

Figure 3.8: kij-Cholesky Factorization : Original Code

for all six versions of Cholesky factorization.

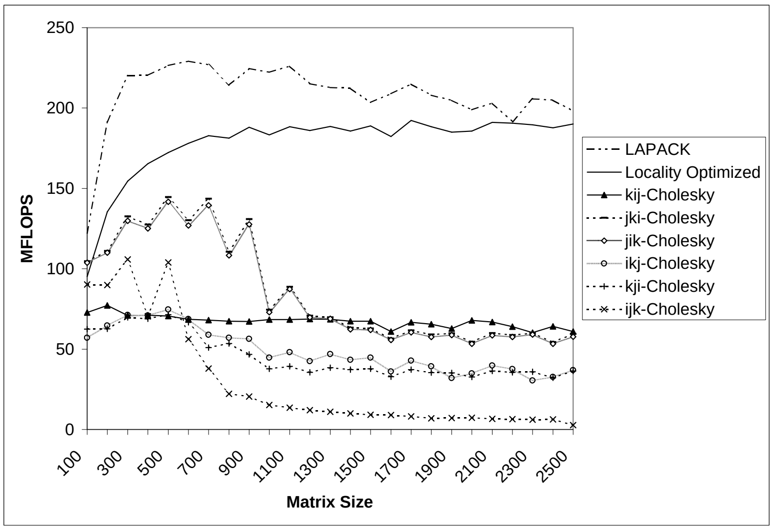
For the *kij* version shown here, the algorithm picks the following embeddings (after reordering and removing redundant dimensions):

$$F_1(\begin{bmatrix} k \end{bmatrix}) = \begin{bmatrix} k \\ k \\ k \end{bmatrix} \qquad F_2(\begin{bmatrix} k \\ i \end{bmatrix}) = \begin{bmatrix} k \\ k \\ i \end{bmatrix} \qquad F_3(\begin{bmatrix} k \\ i \\ j \end{bmatrix}) = \begin{bmatrix} j \\ k \\ i \end{bmatrix}$$

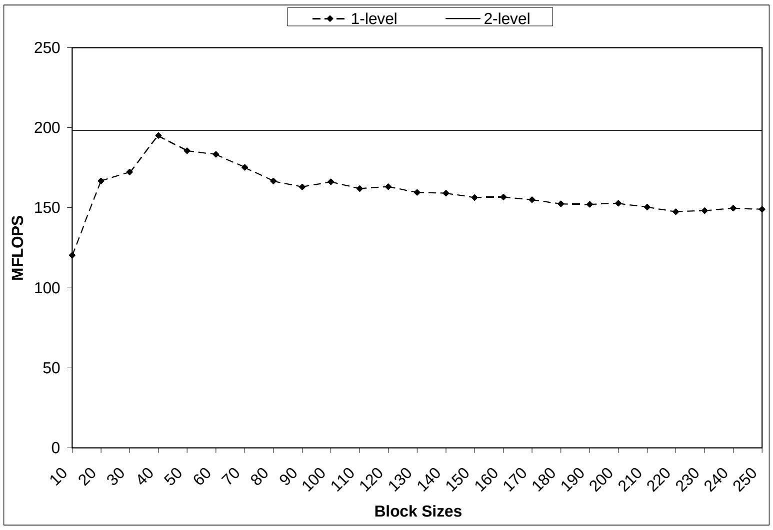
All three dimensions are tiled without skewing. Our algorithm chooses a tile size of 36 for the L1 cache and 288 for the L2 cache for all the dimensions. The same code is obtained starting from any of the six versions of Cholesky factorization, and the line marked "Locality Optimized" in Figure 3.9(c) shows the performance of that code. The code produced by our approach is roughly 3 to 30 times faster than the code produced by the SGI compiler, and it is within 5% of the hand-written LAPACK library code for large matrices. The variation of performance with various tile sizes is shown in Figure 3.9(c) for an array of size 2000×2000.

Jacobi

Our next benchmark is the Jacobi kernel in Figure 3.10. The Jacobi kernel is typical of relaxation codes used to solve *pde*'s using explicit methods. They contain an outer loop that counts time-steps; in each time-step, a stencil computation is performed



(b) Performance



(c) Variation with tile size

Figure 3.9: Cholesky Factorization and its Performance

```
for t = 1,T
    for i = 2,N-1
        for j = 2,N-1
S1:         L(i,j) = (A(i,j+1) + A(i,j-1)
                    + A(i+1,j) + A(i-1,j)) / 4
        end
    end
    for i = 2,N-1
        for j = 2,N-1
S2:         A(i,j) = L(i,j)
        end
    end
end
```

Figure 3.10: Jacobi : Original Code

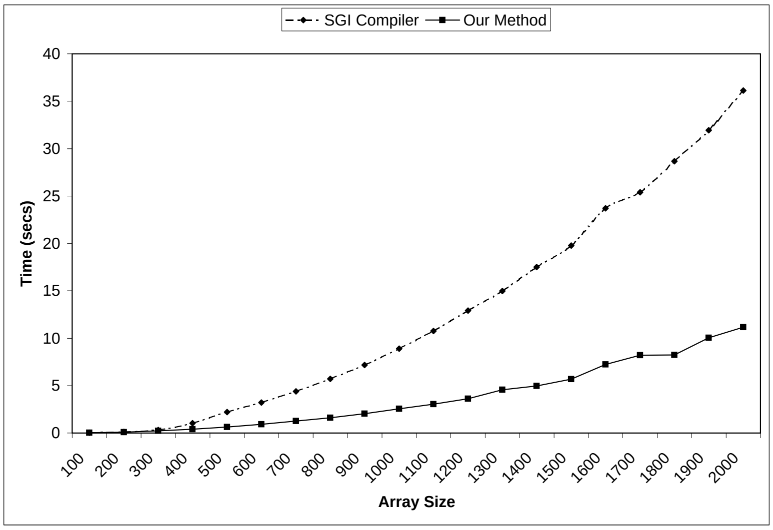
on certain arrays. We show the results of applying our technique to the Jacobi kernel shown in Figure 3.11 which solves Laplace's equation. Our algorithm picks embeddings that perform all the optimization steps discussed in Section 1.

$$F_1(\begin{bmatrix} t \\ i \\ j \end{bmatrix}) = \begin{bmatrix} t \\ j \\ i \end{bmatrix} \quad F_2(\begin{bmatrix} t \\ i \\ j \end{bmatrix}) = \begin{bmatrix} t \\ j+1 \\ i+1 \end{bmatrix}$$

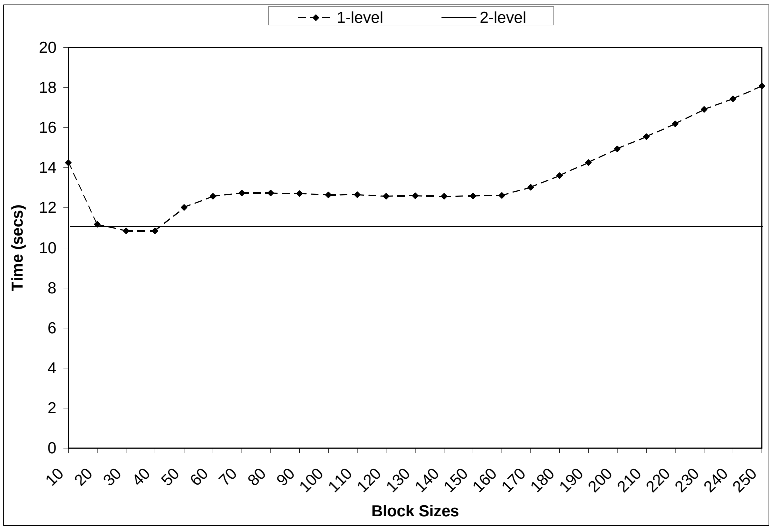
These embeddings correspond to shifting the iterations of the two statements with respect to each other, fusing the resulting `i` and `j` loops respectively, and finally interchanging the `i` and `j` loops. This not only allows us to tile the loops but also benefits the reuses between the two arrays in the two statements, as well as the spatial locality in both statements.

The resulting space cannot be tiled directly, so our implementation chooses to skew the second and the third dimensions by `2*t` before tiling. Our tile size selection algorithm chooses tile sizes of 20 for the L1 cache and 160 for the L2 cache.

Figure 3.11(a) shows the execution times for the code produced by our technique and by the SGI compiler for a fixed number of time-steps (100). As can be seen,



(a) Performance



(b) Variation with tile size

Figure 3.11: Jacobi and its Performance

there is significant performance improvement as a result of the optimizations. The effect of varying tile sizes is shown in Figure 3.11(b).

Red-Black Gauss-Seidel Relaxation

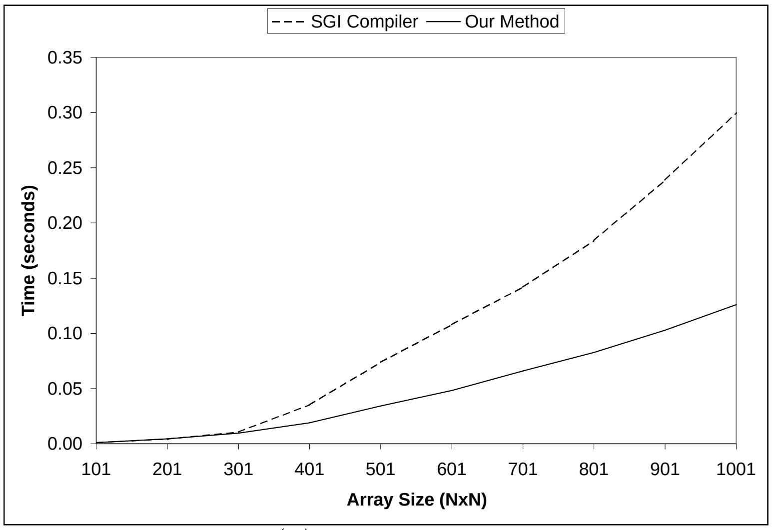
```
for t = 1,T
    for j = 2,N-1
        for i = 2,N-1,2
S1:         U(i,j) = 0.25 * (B(i,j) - U(i-1,j) - U(i+1,j)
                                    - U(i,j+1) - U(i,j-1))
        end
    end
    for j = 2,N-1
        for i = 3,N-1,2
S2:         U(i,j) = 0.25 * (B(i,j) - U(i-1,j) - U(i+1,j)
                                    - U(i,j+1) - U(i,j-1))
        end
    end
end
```

Figure 3.12: Red-Black Gauss Seidel : Original Code

A more complex relaxation code is the Red-Black Gauss-Seidel code used within multi-grid methods to initialize the values for the next grid level. In Figure 3.12, the value of T is typically small (less than 5). The odd and even rows are processed separately. The arrays are touched twice for each time step. Our implementation finds the following embeddings:

$$F_1(\begin{bmatrix} t \\ j \\ i \end{bmatrix}) = \begin{bmatrix} t \\ j \\ i \end{bmatrix} \quad F_2(\begin{bmatrix} t \\ j \\ i \end{bmatrix}) = \begin{bmatrix} t \\ j \\ i+1 \end{bmatrix}$$

These embeddings effectively fuse the odd and even loops together thereby cutting down the memory traffic by a factor of two. The last three dimensions are redundant. The code can be tiled after skewing both the `i` and `j` loops by `1*t`. Our tile size selection algorithm chooses 28 for the L1 cache and 224 for the L2 cache. Figure 3.13 shows the performance of the resulting code when the number of time-steps is set to 5.



(b) Performance

Figure 3.13: Red-Black Gauss-Seidel and its Performance

Tomcatv

As a final example, we consider the tomcatv code from the SPECfp benchmark suite. The code (Figure 3.14) consists of an outer time loop `ITER` containing a sequence of doubly- and singly-nested loops which walk over both two-dimensional and one-dimensional arrays. The results of applying our technique are shown in Figure 3.16(a) for a fixed array size (253 from a reference input), and a varying number of time-steps. Tomcatv is not directly amenable to our technique because it contains an exit test at the end of each time-step. The line marked "Locality Optimized" represents the results of optimizing a single time-step (i.e. the code inside the `ITER` loop) for locality. Treating every basic block as a single statement, our algorithm produces an embedding which corresponds to fusing some of the `J` loops and all the `I` loops. The exploitation of reuse between different basic blocks results in roughly 8% improvement in performance compared to the code produced by the SGI compiler. If we consider the tomcatv kernel without the exit condition[2], our algorithm skews the fused `I` loop by `2*ITER`, and then tiles `ITER` and the skewed `I` loops. Our algorithm decides to tile only for the L2 cache (the data touched by a tile does not fit into L1 cache) with a tile size of 48.

The performance of the resulting code (line marked "Tiled") is around 22% better than the original code. Variation with tile size is shown in Figure 3.16(b).

[2] The resulting kernel can be tiled speculatively as demonstrated by Song and Li [31].

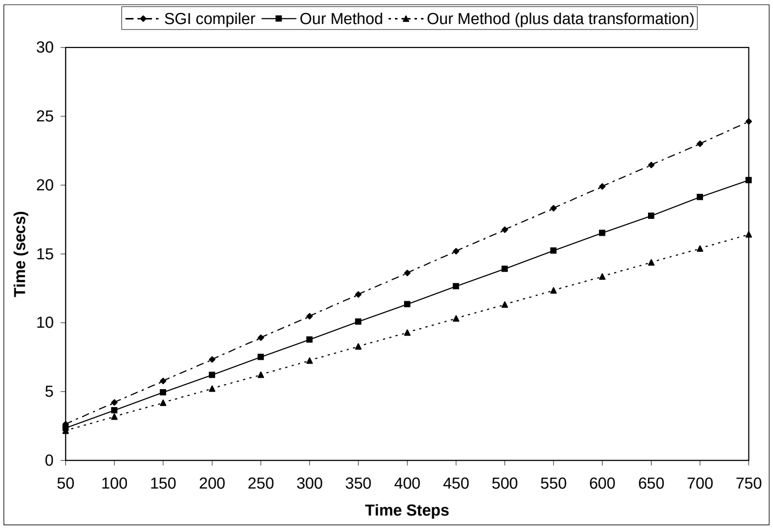
```
DO      140      ITER = 1, ITACT
C
C     Residuals of ITER iteration
C
        RXM(ITER)  = 0.D0
        RYM(ITER)  = 0.D0
C
        DO     60    J = 2,N-1
C
          DO     50    I = 2,N-1
            XX = X(I+1,J)-X(I-1,J)
            YX = Y(I+1,J)-Y(I-1,J)
            XY = X(I,J+1)-X(I,J-1)
            YY = Y(I,J+1)-Y(I,J-1)
            A  = 0.25D0  * (XY*XY+YY*YY)
            B  = 0.25D0  * (XX*XX+YX*YX)
            C  = 0.125D0 * (XX*XY+YX*YY)
            AA(I,J) = -B
            DD(I,J) = B+B+A*REL
            PXX = X(I+1,J)-2.D0*X(I,J)+X(I-1,J)
            QXX = Y(I+1,J)-2.D0*Y(I,J)+Y(I-1,J)
            PYY = X(I,J+1)-2.D0*X(I,J)+X(I,J-1)
            QYY = Y(I,J+1)-2.D0*Y(I,J)+Y(I,J-1)
            PXY = X(I+1,J+1)-X(I+1,J-1)-X(I-1,J+1)+X(I-1,J-1)
            QXY = Y(I+1,J+1)-Y(I+1,J-1)-Y(I-1,J+1)+Y(I-1,J-1)
C
C     CALCULATE RESIDUALS ( EQUAL TO RIGHT HAND SIDES OF EQUS.)
C
            RX(I,J)   = A*PXX+B*PYY-C*PXY
            RY(I,J)   = A*QXX+B*QYY-C*QXY
C
   50     CONTINUE
   60   CONTINUE
C
C     DETERMINE MAXIMUM VALUES RXM, RYM OF RESIDUALS
C
        DO     80    J = 2,N-1
          DO     80    I = 2,N-1
            RXM(ITER) = MAX(RXM(ITER), ABS(RX(I,J)))
            RYM(ITER) = MAX(RYM(ITER), ABS(RY(I,J)))
   80   CONTINUE
C
C     CONTINUED ON NEXT PAGE...
C
```

Figure 3.14: Tomcatv Kernel

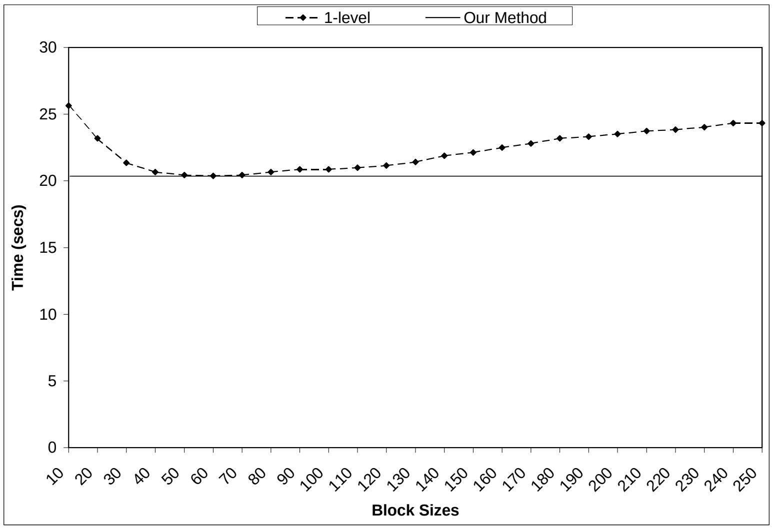
```
C
C     SOLVE TRIDIAGONAL SYSTEMS (AA,DD,AA) IN PARALLEL, LU DECOMPOSITION
C
        DO     90    I = 2,N-1
          D(I,2) = 1.D0/DD(I,2)
   90   CONTINUE
        DO    100     J = 3,N-1
          DO    100     I = 2,N-1
            R       = AA(I,J)*D(I,J-1)
            D (I,J) = 1.D0/(DD(I,J)-AA(I,J-1)*R)
            RX(I,J) = RX(I,J) - RX(I,J-1)*R
            RY(I,J) = RY(I,J) - RY(I,J-1)*R
  100   CONTINUE
        DO    110     I = 2,N-1
          RX(I,N-1) = RX(I,N-1)*D(I,N-1)
          RY(I,N-1) = RY(I,N-1)*D(I,N-1)
  110   CONTINUE
        DO    120     J = N-2,2,-1
          DO    120     I = 2,N-1
            RX(I,J) = (RX(I,J)-AA(I,J)*RX(I,J+1))*D(I,J)
            RY(I,J) = (RY(I,J)-AA(I,J)*RY(I,J+1))*D(I,J)
  120   CONTINUE
C
C     ADD CORRECTIONS OF ITER ITERATION
C
        DO    130     J = 2,N-1
          DO    130     I = 2,N-1
            X(I,J) = X(I,J)+RX(I,J)
            Y(I,J) = Y(I,J)+RY(I,J)
  130   CONTINUE
C
        ABX  = ABS(RXM(ITER))
        ABY  = ABS(RYM(ITER))
        IF (ABX.LE.EPS.AND.ABY.LE.EPS)  GOTO  150
  140 CONTINUE
C
C     END OF ITERATION LOOP 14

  150 CONTINUE
```

Figure 3.15: Tomcatv Kernel (continued)



(a) Performance



(b) Variation with tile size

Figure 3.16: Performance of Tomcatv

Chapter 4

Generating Block-recursive Codes

Modern processor architectures have multiple levels of memory hierarchy. For example, current processors like Intel's Merced have three levels of caches. Current compilers optimize for these levels by tiling loop-nests separately for each level of the hierarchy.

In the dense numerical linear algebra community, there is growing interest in the use of block-recursive versions of numerical kernels like matrix multiply and Cholesky factorization to address the same problem. These algorithms recursively partition the original problem into ones with smaller working sets. This recursion has the effect of blocking the data at many different levels at the same time, so the data access patterns of the resulting codes exploit locality at all levels of the memory hierarchy. Experiments by Gustavson [15] and others have shown that these algorithms achieve substantial performance improvement over the tiled versions of the codes. In this chapter, we show how to use the product space framework to automatically generate the block-recursive versions of these numerical kernels. We will use as examples block-recursive codes for two algorithms: matrix multiplication and Cholesky factorization.

```
for j = 1, n
    for k = 1, j-1
        for i = j, n
S1:         A(i,j) -= A(i,k) * A(j,k)
        end
    end
S2: A(j,j) = dsqrt(A(j,j))
    for i = j+1, n
S3:     A(i,j) = A(i,j) / A(j,j)
    end
end
```

Figure 4.1: Cholesky Factorization

Consider the code fragment shown in Figure 4.1. It is an *iterative* version of Cholesky factorization that factories a symmetric positive definite matrix `A` such that $A = L \cdot L^T$ where L is a lower triangular matrix. In the code fragment shown here, the matrix L overwrites A. A block-recursive version of the algorithm can be obtained by sub-dividing the arrays A and L into 2×2 blocks and equating terms on both sides.

$$\begin{bmatrix} A_{00} & A_{10}^T \\ A_{10} & A_{11} \end{bmatrix} = \begin{bmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{bmatrix} \begin{bmatrix} L_{00}^T & L_{10}^T \\ 0 & L_{11}^T \end{bmatrix} = \begin{bmatrix} L_{00}L_{00}^T & L_{00}L_{10}^T \\ L_{10}L_{00}^T & L_{10}L_{10}^T + L_{11}L_{11}^T \end{bmatrix}$$

$$\begin{aligned} L_{00} &= chol(A_{00}) \\ L_{10} &= A_{10}L_{00}^{-T} \\ L_{11} &= chol(A_{11} - L_{10}L_{10}^T) \end{aligned}$$

Here $chol(X)$ computes the Cholesky factorization of array X. The recursive version performs a Cholesky factorization of the A_{00} block, then a division on the A_{10} block, and finally performs another Cholesky factorization on the updated A_{11} block. The termination condition for the recursion can either be a single element of A (degenerating to square root operation) or to a $b \times b$ block of A which can be solved by the iterative code fragment.

```
for j = 1, n
    for k = 1, n
        for i = 1, n
            C(i,j) -= A(i,k) * B(k,j)
        end
    end
end
```

Figure 4.2: Matrix Multiplication

A recursive version of matrix multiplication $C = AB$ can also be derived in a similar manner. The iterative code is shown in Figure 4.2. Subdividing the arrays into 2×2 blocks results in the following :-

$$\begin{aligned}
\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} &= \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} \\
&= \begin{bmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{bmatrix}
\end{aligned}$$

In the above formulation, each multiply results in eight recursive calls acting on sub-blocks, with 2 recursive calls per sub-block. The 4 sub-blocks of C can be traversed in any order. The natural order of traversing the blocks of C is the lexicographic order. If this order is applied recursively, we obtain the *block-recursive* order of traversing the two-dimensional space represented by the matrix C. This is shown in Figure 4.4. This manner of walking would correspond to the ordering of the recursive calls shown in Figure 4.3. Note that we can also consider traversing the sub-blocks of matrix A or B in a similar manner.

The ordering shown in Figure 4.3 is not the only way of ordering the recursive calls. In fact, unlike the Cholesky factorization case, the eight recursive calls in matrix multiplication can be performed in any order as they are all independent. One way of ordering these calls is to make sure that one of the operands is reused

$$MMM(C_{00}, A_{00}, B_{00})$$
$$MMM(C_{00}, A_{01}, B_{10})$$
$$MMM(C_{01}, A_{00}, B_{01})$$
$$MMM(C_{01}, A_{01}, B_{11})$$
$$MMM(C_{10}, A_{10}, B_{00})$$
$$MMM(C_{10}, A_{11}, B_{10})$$
$$MMM(C_{11}, A_{10}, B_{01})$$
$$MMM(C_{11}, A_{11}, B_{11})$$

Figure 4.3: Block-recursive call order

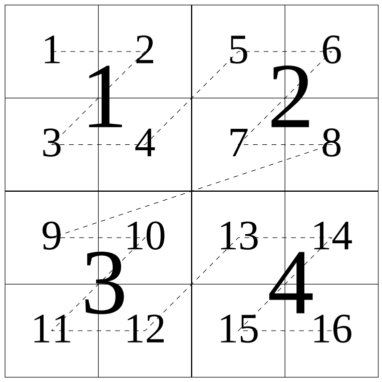


Figure 4.4: Block Recursive Order

between adjacent calls[1]. The ordering corresponding to block-recursive traversals does not have this property – for example, no blocks are reused between the second and the third recursive calls in Figure 4.3.

An ordering of the calls satisfying the reuse property is shown in Figure 4.5. This corresponds to traversing the sub-blocks of C in a *gray-code* order. A gray-code order on the set of numbers $(1 \ldots m)$ arranges the numbers so that adjacent numbers differ by exactly 1 bit in their binary representation.

A gray-code order of traversing a 2-dimensional space is shown in Figure 4.6. Such an order is called *space-filling*, since the order traces a complete path through all the points, always moving from one point to an adjacent point. There are other space-filling orders, and some of them are described in the references [7].

For comparison, the lexicographic order of traversing a two-dimensional space is shown in Figure 4.7. Note that neither the lexicographic order nor the block-recursive order are space-filling orders.

[1]Not more than one can be reused, in any case.

$$MMM(C_{00}, A_{00}, B_{00})$$
$$MMM(C_{00}, A_{01}, B_{10})$$
$$MMM(C_{01}, A_{01}, B_{11})$$
$$MMM(C_{01}, A_{00}, B_{01})$$
$$MMM(C_{11}, A_{10}, B_{01})$$
$$MMM(C_{11}, A_{11}, B_{11})$$
$$MMM(C_{10}, A_{11}, B_{10})$$
$$MMM(C_{10}, A_{10}, B_{00})$$

Figure 4.5: Gray-code call order

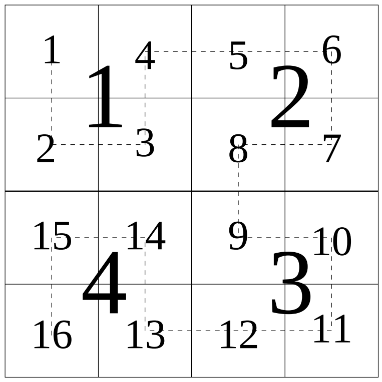


Figure 4.6: Space-Filling Order

This chapter describes compiler technology that can automatically convert iterative versions of array programs into their recursive versions. In these programs, arrays are referenced via affine functions of the loop-index variables. As a result, partitioning the iterations of a loop will result in the partitioning of data as well. We exploit this insight as follows. We use affine mapping functions to map all the statement instances of the program to the *product space*. As we have seen already, this mapping effectively converts the program into a perfectly-nested loop-nest, with all statements nested in the innermost loop. We develop legality conditions under which the loops (which correspond to the dimensions of the space) can be recursively bisectioned. This corresponds to recursively bisectioning the dimensions of the program iteration space. Code is then generated to traverse the space in a block-recursive or space-filling manner, and when each point in this space is visited, the statements mapped to it are executed. This strategy effectively converts the iterative versions of codes into their recursive ones. The mapping functions that enable this conversion can be automatically derived when they exist.

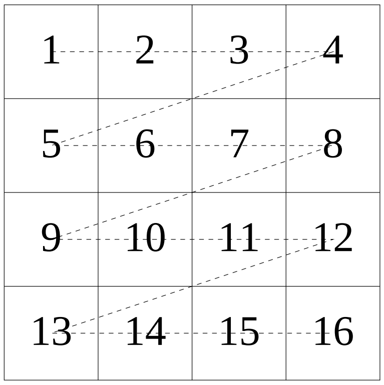


Figure 4.7: Lexicographic Order

4.1 The Product Space

As in the previous chapters, we consider a program to consist of statements contained within loops. All loop bounds and array access functions are assumed to be affine functions of surrounding loop indices. We will use S_1, S_2, $\ldots$, S_n to name the statements of the program in syntactic order.

Consider a legal program execution order for the above program given by the pair $(\mathcal{P}, \mathcal{F} = \{F_1, F_2, \ldots, F_n\})$. Here $\mathcal{P}$ is the product space and $\mathcal{F}$ is the set of functions that map dynamic statement instances to the product space. Since the above execution order is legal, legal code can be generated by traversing the program iteration space lexicographically and executing statement instances mapped to each point when the point is visited. Note that the lexicographic traversal may not be the only legal order of traversing the program iteration space. In particular, other legal orders of traversing the space can be obtained in the following ways.

Consider the set of pairs (i_s, i_d) for the program such that a dependence exists from statement instance $S_s(i_s)$ to instance $S_d(i_d)$. The set $v = F_d(i_d) - F_s(i_s)$ is the set of difference vectors associated with the program.

1. Any order of walking the product space represented by a unimodular transformation matrix T is legal if $T \cdot v$ is lexicographically positive for every difference vector v associated with the code.

2. As discussed in Chapter 3.1, if the entries of all difference vectors corresponding to a set of dimensions of the product space are **non-negative**, then those dimensions can be blocked. This partitions the product space into blocks with planes parallel to the axes of the dimensions. These blocks are visited in lexicographic order. When a particular block is visited, all points within that block are visited in lexicographic order as well. This order of traversal for a two-dimensional product space divided into equal-sized blocks is shown in Figure 4.7.

   Note that the points within each block do not need to be visited lexicographically. Any set of dimensions which can be blocked can also be *recursively blocked* i.e. each block can itself be further blocked and these inner blocks can either be traversed lexicographically or be further blocked recursively. If we choose to block the program iteration space by recursively bisecting block-dimensions then we obtain the block-recursive order shown in Figure 4.4.

3. If the entries corresponding to a particular dimension of the product space are **zero** for all difference vectors, then that dimension does not have to be traversed lexicographically — it can be traversed in any order. If a set of dimensions exhibit this property, then not only can those dimensions be blocked, but the blocks themselves do not have to be visited in a lexicographic order. In particular, the blocks of these dimensions can be traversed in a *space-filling* order. This principle can be applied recursively within each block, to obtain

space-filling orders of traversing the entire sub-space (Figure 4.6).

Given an execution order $(\mathcal{P}, \mathcal{F})$, and the dependences in the program, it is easy to check if the difference vectors exhibit the above properties using standard dependence analysis [26]. If we limit our embedding functions $\mathcal{F}$ to be affine functions of the loop-index variables and symbolic constants, we can determine functions which allow us to block dimensions (and hence also recursively block them) or to traverse a set of dimensions in a space-filling order. The condition that entries corresponding to a particular dimension of all difference vectors must be non-negative (for recursive-blocking) or zero (for space-filling orders) can be converted into a system of linear inequalities on the unknown coefficients of $\mathcal{F}$ by an application of *Farkas' Lemma* as discussed in Section 2.4.1. If this system has solutions, then any solution satisfying the linear inequalities would give the required embedding functions.

4.2 Code Generation

Consider an execution order of a program represented by the pair $(\mathcal{P}, \mathcal{F})$. Let p represent the number of dimensions in the product-space. We wish to block the program iteration space recursively, terminating when blocks of size $B \times B \ldots \times B$ are reached.

For simplicity we will assume that redundant dimensions have been removed and that all dimensions can be blocked. We will also assume that all points in the program iteration space that have statement instances mapped to them are positive and that they are all contained in the bounding box $(1 \cdots B \times 2^{k_1}, \ldots, 1 \cdots B \times 2^{k_p})$. This can be ensured by choosing suitably large values of $k_1, \ldots k_p$.

Code to recursively traverse the product-space is shown in Figure 4.8. The procedure `Recurse` is parameterized with the current block to traverse given by `(lb[1]:ub[1], ... , lb[p]:ub[p])`. If the current block is not the base block (the termination condition), `GenerateRecursiveCalls` subdivides the block into 2^p sub-blocks by bisecting each dimension and calls `Recurse` recursively in a lexicographic order[2]. If the termination condition is reached, code for the block is executed in `BlockCode`. The function `HasPoints` prevents the code from recursing into blocks that have no statement instances mapped to them. The initial call to `Recurse` is made with the lower and upper bounds set to the bounding box.

Naive code for `BlockCode(lb,ub)` is very similar to the naive code for executing the program. Instead of traversing the entire product space we only need to traverse the points in the current block lexicographically, and execute statement instances mapped to them. The redundant loops and conditionals can be hoisted out by employing polyhedral techniques.

We can identify blocks `(lb[1]:ub[1], ... ,lb[p]:ub[p])` that contain points with statement instances mapped to them by creating a linear system of inequalities with variables lb_i, ub_i corresponding to each entry of `lb[1..p]`, `ub[1..p]` and variables x_i corresponding to each dimension of the product-space. Constraints are added to ensure that the point $(x_1, x_2, \ldots, x_p)$ has a statement instance mapped to it and that it lies within the block $(lb_1 : ub_1, \ldots, lb_p : ub_p)$. From the above system, we obtain the condition to be tested in `HasPoints(lb,ub)` by projecting out (in the Fourier-Motzkin sense) the variables x_i.

[2] This must be changed appropriately if space-filling orders are required

```
Recurse(lb[1..p], ub[1..p])
if (HasPoints(lb,ub)) then
    if (∀i ub[i] == lb[i]+B-1) then
        BlockCode(lb)
    else
        GenerateRecursiveCalls(lb,ub,1)
   endif
endif
end


GenerateRecursiveCalls(lb[1..p], ub[1..p], q)
if (q > p)
    Recurse(lb, ub)
else
  for i = 1,p
      lb'[i] = lb[i]
      ub'[i] = (i == q) ? (lb[i]+ub[i])/2 : ub[i]
  endfor
  GenerateRecursiveCalls(lb',ub',q+1)

  for i = 1, p
      lb'[i] = (i == q) ? (lb[i]+ub[i])/2 + 1 : lb[i]
      ub'[i] = ub[i]
  endfor
  GenerateRecursiveCalls(lb',ub',q+1)
endif
end
```

Figure 4.8: Recursive code generation

$$F_1\left(\begin{bmatrix} j_1 \\ k_1 \\ i_1 \end{bmatrix}\right) = \begin{bmatrix} j_1 \\ k_1 \\ i_1 \\ j_1 \\ j_1 \\ i_1 \end{bmatrix} \quad F_2\left(\begin{bmatrix} j_2 \end{bmatrix}\right) = \begin{bmatrix} j_2 \\ j_2 \\ j_2 \\ j_2 \\ j_2 \\ j_2 \end{bmatrix} \quad F_3\left(\begin{bmatrix} j_3 \\ i_3 \end{bmatrix}\right) = \begin{bmatrix} j_3 \\ j_3 \\ i_3 \\ j_3 \\ j_3 \\ i_3 \end{bmatrix}$$

Figure 4.9: Embeddings for Cholesky

4.3 Examples

4.3.1 Cholesky Factorization

For our Cholesky example, the embedding functions shown in Figure 4.9 allow all dimensions to be blocked. Since there are difference vectors with non-zero entries, the program iteration space cannot be walked in a space-filling manner, though it can be recursively blocked. The naive code for executing the code in each block is shown in Figure 4.10. This code traverses the current block of the product space lexicographically and when a point is visited, all statement instances mapped to it are executed in original program order. As mentioned earlier, the redundant loops must be removed and the conditionals hoisted out for good performance. The part of the product-space that has statement instances mapped to it is $[j, k, i] : 1 \leq k \leq j \leq i \leq n$. This is used to obtain the condition in `HasPoints()`.

4.3.2 Matrix Multiplication

For matrix multiplication, the embeddings shown in Figure 4.12 not only allow recursive blocking but also allow the product space to be traversed in any space-

```
BlockCode(lb[1..3])
for j1 = lb[1], lb[1]+B-1
for k1 = lb[2], lb[2]+B-1
for i1 = lb[3], lb[3]+B-1
    for j = 1, n
        for k = 1, j-1
            for i = j, n
                if (j1==j && k1==k && i1==i)
S1:                 A(i,j) -= A(i,k) * A(j,k)
                endif
            end
        end
        if (j1==j && k1==j && i1==j)
S2:         A(j,j) = dsqrt(A(j,j))
        endif
        for i = j+1, n
            if (j1==j && k1==j && i1==i)
S3:             A(i,j) = A(i,j) / A(j,j)
            endif
        end
    end
end
end
end


HasPoints(lb[1..3], ub[1..3])
if (lb[1]<=n && lb[2]<=n && lb[3]<=n
    && lb[1]<=ub[3]
    && lb[2]<=ub[1]
    && lb[2]<=ub[3])
    return true
else
    return false
endif
```

Figure 4.10: Recursive code for Cholesky

```
BlockCode(lb[1..3])
for j1 = lb[1], lb[1]+B-1
for k1 = lb[2], lb[2]+B-1
for i1 = lb[3], lb[3]+B-1
    for j = 1, n
        for k = 1, n
            for i = 1, n
                if (j1==j && k1==k && i1==i)
S1:                 C(i,j) += A(i,k) * B(k, j)
                endif
            end
        end
    end
end
end
end


HasPoints(lb[1..3], ub[1..3])
if (lb[1]<=n && lb[2]<=n && lb[3]<=n)
    return true
else
    return false
endif
```

Figure 4.11: Recursive code for Matrix Multiplication

filling manner. The naive code to execute the statements mapped to a base block is shown in 4.11. `HasPoint()` just ensures that there are statements mapped to the base block.

4.4 Experimental Results

In this section, we discuss the performance of block-recursive and space-filling codes produced using the technology described in this paper. The legality conditions discussed in Section 4.1 allow us to decide that the matrix multiply example (MMM, Figure 4.2) can be blocked both recursively as well as in a space-filling manner. The

$$F_1(\begin{bmatrix} j_1 \\ k_1 \\ i_1 \end{bmatrix}) = \begin{bmatrix} j_1 \\ k_1 \\ i_1 \end{bmatrix}$$

Figure 4.12: Embeddings for Matrix Multiplication

Cholesky code in Figure 4.1 can only be blocked recursively. We generated recursive code terminating in four different base block sizes (16, 32, 64, 128) for both programs. The codes for these base block sizes (`BlockCode(lb)`) were compiled with the "-O3 -LNO:blocking=off" option of the SGI compiler. At this level of optimization, the SGI compiler performs tiling for registers and software-pipelining.

For each program, we ran the recursive (and if legal, the space-filling) versions of the code for a variety of matrix sizes For lack of space, we will only present results for a matrix size of 4000×4000. Results for other matrix sizes are similar. In the graphs, the results marked `Lexicographic` correspond to executing the code in `BlockCode(lb)` by visiting the base blocks in a lexicographic manner. This has the effect of blocking (tiling) the program iteration space. We also show results of executing vendor-supplied hand-tuned implementations of matrix multiply (`BLAS`) and Cholesky (`LAPACK` [2]), for comparison. All experiments were run on an SGI R12K machine running at 300Mhz with a 32Kb primary-data cache (L1), 2Mb second-level cache (L2) and 64 TLB entries.

4.4.1 Overheads

Figures 4.13 and 4.14 show the overhead in the generated recursive versions of matrix multiplication and Cholesky kernels. These experiments were run on arrays of size 1000×1000 for various base block sizes ranging from 1 to 256. The size 1000

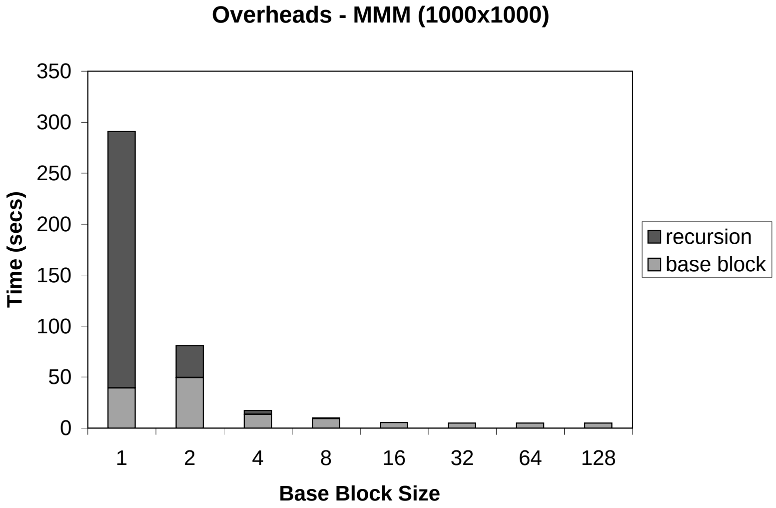


Figure 4.13: MMM : Overheads

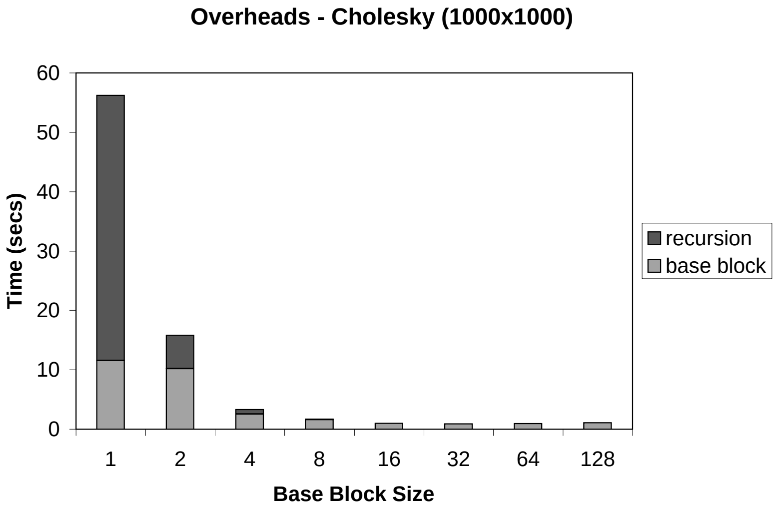


Figure 4.14: CHOL : Overheads

was chosen so that the arrays would fit into the second level cache and therefore not incur too many cache misses. The dark grey portions of the execution time represent the time spent in the recursion control structure while the light grey portions represent the actual time spent in the base block code. As can be seen from the figures, overheads in these codes are from two different sources –

1. Recursion Overheads : These overheads are strictly due to the recursion con-

trol flow structure. They are represented by the dark gray portions of the execution time in the figures. As can be seen, recursion overhead is huge for very small base block sizes. As the base block size increase, the recursion depth decreases resulting in a decrease in the recursion overhead. It accounts for less than 1% of the execution time for base block sizes greater than 16.

2. Base block code Overheads : For very small base block sizes, the backend of the compiler is not able to generate good low level code as it cannot take advantage of software pipelining, and register allocation strategies. This results in the difference in the time spent in the base block codes for various base block sizes. As can be seen, base block sizes of upto 16 do not provide enough instructions for efficient scheduling by the backend.

4.4.2 Memory Hierarchy Performance

Multiple Cache Levels

Figures 4.15 and 4.16 show the number of primary data cache misses for the two programs. For the larger block sizes (64, 128), the data touched by a base block does not fit into cache (32K) and hence both the recursive and lexicographic versions suffer the same penalty. For smaller block sizes (16, 32), the data fits into cache resulting in much fewer misses. There is a small difference between the lexicographic and recursive versions for a block size of 16. The lexicographic versions have slightly more misses than the recursive versions since at this block size, the data touched by a base block fits into less than 25% of the first-level cache. The recursive doubling effect aids the recursive-versions in using the cache more effectively. This effect is more pronounced for the second-level cache misses shown in Figures 4.17 and 4.18.

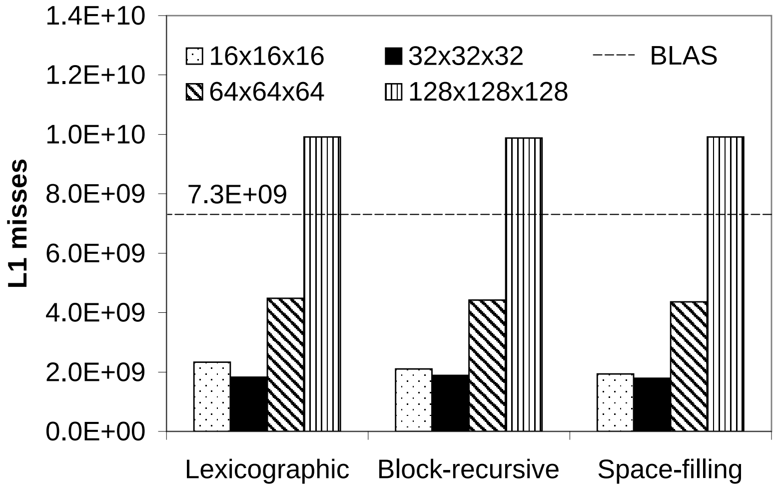


Figure 4.15: MMM : L1 misses

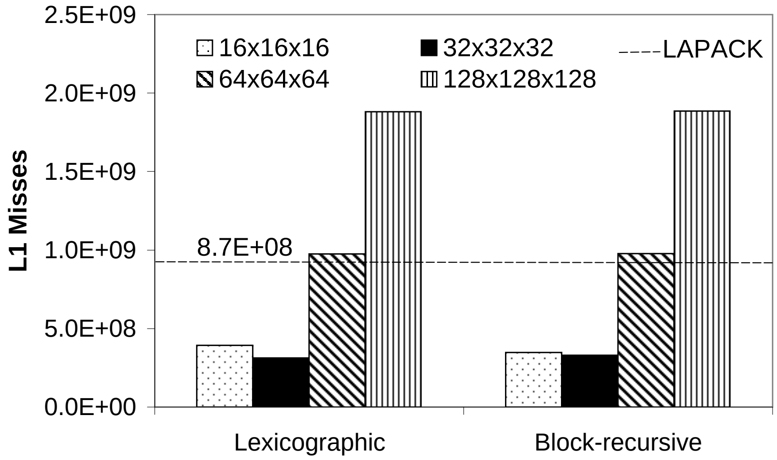


Figure 4.16: CHOL : L1 misses

The lexicographic versions for block sizes of 16 and 32 exhibit much higher miss numbers than the corresponding recursive versions since these block sizes are too small to fully utilize the 4M cache. In the recursive versions, however, even the small block sizes succeed in full utilization of the cache. These recursive versions will have a similar effect on any further levels of caches. Of the two recursive orders, the space-filling orders show slightly better cache performance for both programs.

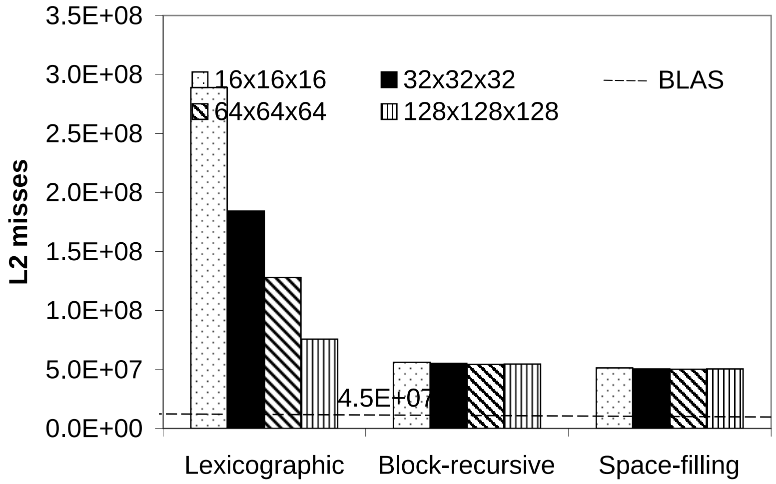


Figure 4.17: MMM : L2 misses

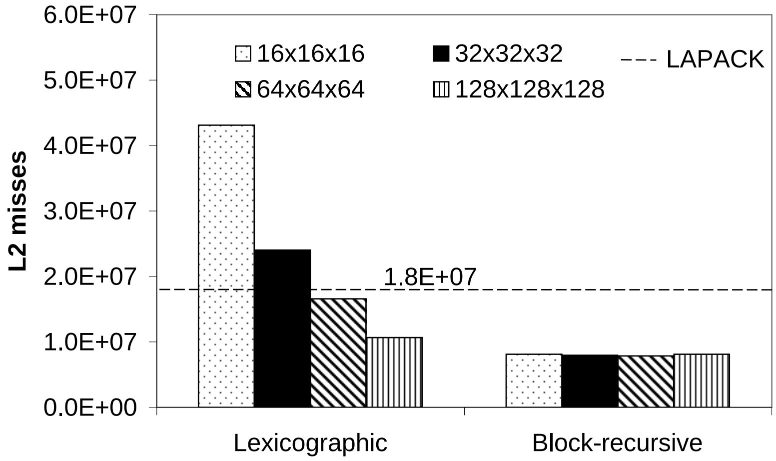


Figure 4.18: CHOL : L2 misses

TLB performance

Figures 4.19 and 4.20 show the number of TLB misses for the two programs. The R12K TLB has only 64 entries, hence large block sizes (more than 64) will exhibit high miss rates in both the lexicographic and recursive cases. Small block sizes could work well in the lexicographic case if the loop-order is chosen well. In our case, the `jki`-order is the best order for both the programs, and hence the case when block size is 16 has very few TLB misses as it uses less than 48 entries at a time. In the

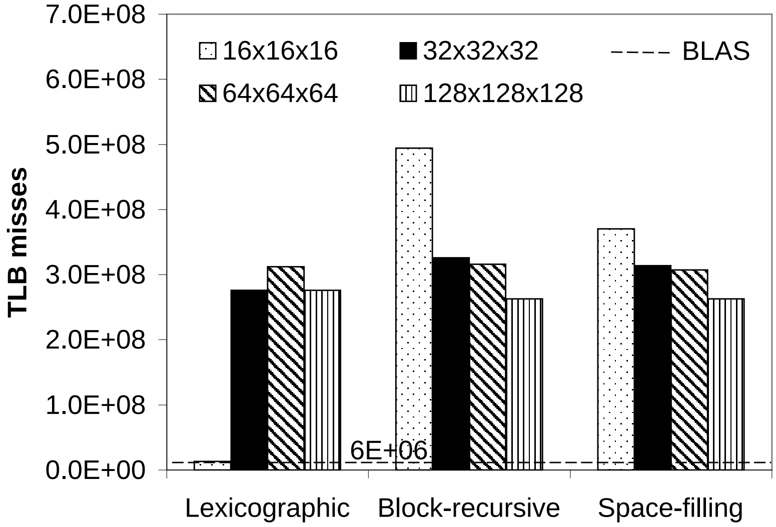


Figure 4.19: MMM : TLB misses

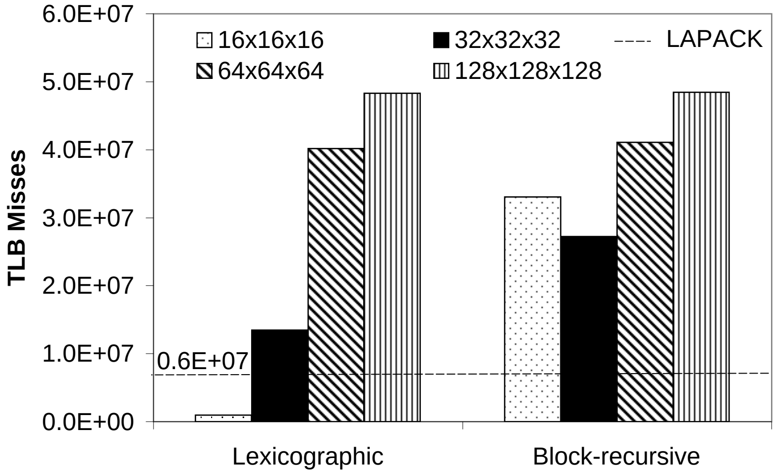


Figure 4.20: CHOL : TLB misses

recursive case, the recursive doubling does cause significantly more TLB misses for small block sizes, although the recursive walks are largely immune to the effect of reordering the loops. In comparison, in the `jik`-order (not shown here), the code with a base block size of 16 suffers a 100-fold increase in the number of TLB misses for the lexicographic case but remains roughly the same in the recursive cases.

4.4.3 Discussion

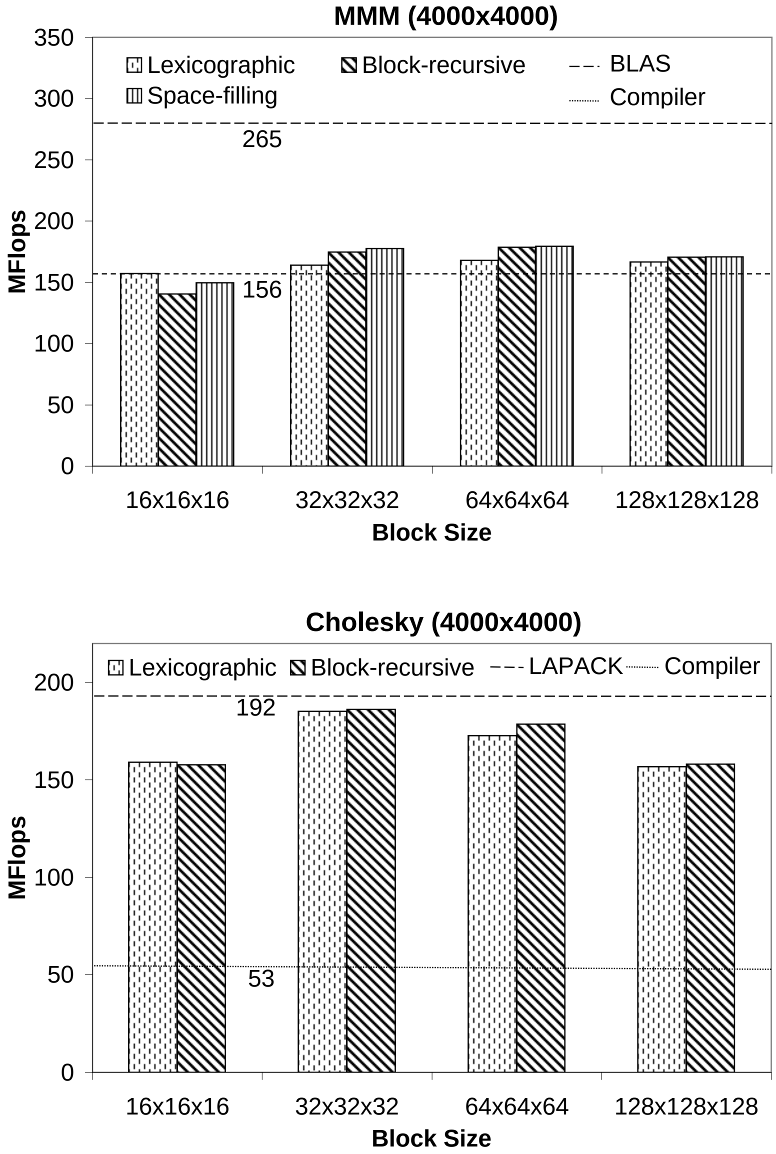


Figure 4.21: Performance

Figure 4.21 shows the performance of the two programs in MFlops. As a sanity check, the lines marked `Compiler` show the performance obtained with compiling the original code with the "-O3" flag of the SGI compiler which attempts to tile for cache and registers and then software-pipeline the resulting code. For both programs, the recursive codes with block size of 32 are the best among all the generated code. For most block sizes, the recursive codes are better than their lexicographic counterparts by a small percentage (2-5%). This is not the case when the block size is 16 because of the large number of TLB misses in the recursive cases for this block size.

For matrix multiply, the best recursive code generated by the compiler is still substantially worse than the hand-tuned versions of the programs even though the recursive overhead is less than 1% in all cases. This difference could be due to the high number of TLB misses suffered by the recursive versions. Better interaction with the TLB requires either (i) copying data from column-major order into recursive data layouts as suggested by Chatterjee [7] or (ii) copying the data used by a base block into contiguous locations as suggested by Gustavson [29]. It is also interesting to note that although the hand-tuned version suffers higher primary cache miss rates, the impact on performance is small. This is not surprising in an out-of-order issue processor like the R12K where the latency of primary cache misses (10 cycles) can be hidden by scheduling and software-pipelining. These misses will be more important in an in-order issue processor like the Merced. For Cholesky factorization, on the other hand, the best block-recursive version is comparable in performance to LAPACK code.

Chapter 5

Conclusions

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

5.1 Summary of Dissertation

As the difference in speed between processors and memory increases, it becomes crucial to take advantage of the locality of reference present in a program and make effective use of caches. Most programs are not written to take advantage of the locality inherent in the algorithm. Discovering this locality and writing the code so it takes advantage of it is not an easy task. These optimized programs are not easy to write – they are very complicated, requiring a wide variety of program transformations and the determination of important parameters that vary from machine architecture to architecture.

One approach of tackling this problem is to develop efficient libraries. Certain core computations can be developed as a library for each architecture. This is the case with libraries like `BLAS` (Basic Linear Algebra Subroutines). This library contains core subroutines like matrix multiplication and matrix vector products that are present in linear algebra code. These routines are hand-written and well-tuned for a particular architecture by the vendors of that architecture. Other applications and libraries can then be written in terms of these routines. For example, the `LAPACK` library is a portable library that provides a wide range of linear algebra algorithms that have been restructured to take advantage of the BLAS subroutines. The main disadvantage of this approach that it is limited in it's applicability and restructuring an algorithm to take advantage of the provided routines may not be easy. Further, these libraries require an enormous amount of development effort that must be repeated for every new architecture.

An alternate method has been advocated by compiler writers. Compiler writers have proposed the automatic restructuring of programs so that the final programs are more cache-efficient. A large number of program transformations have been studied and much research has been undertaken to discover when these transformations can be legally applied and how to decide which transformations should be used and in which sequence they should be applied. This approach has been very successful for a certain class of codes in which all memory accesses are through scalar variables or arrays and all statements are contained in the innermost loop. Such programs are called *perfectly-nested*. Such codes can be optimized using a linear loop transformation framework. Two key transformations – loop permutation and tiling – are used to convert such codes into semantically equivalent versions that use the caches more effectively.

Unfortunately, this technology cannot be applied to the majority of codes which are imperfectly-nested. Various ad-hoc strategies have been used in practice to convert imperfectly-nested loops into perfectly-nested via code transformations like code-sinking, fusion and fission. This introduces a phase ordering problem – no known technique is known which chooses the correct sequence of transformation in all cases. Special purpose techniques that rely on the structure of the code have been suggested for matrix factorization codes and relaxation codes.

In my thesis, I propose an approach that tackles the issue of determining the transformations that need to be applied in order to enhance locality in imperfectly-nested codes. This technology generalizes the concepts introduced for the perfectly-nested case. In particular, the iteration space of an imperfectly-nested loop nest is modeled by an integer lattice called the *product space* as discussed in Chapter 2.3.2. The product space is the Cartesian product of the iteration spaces of the individual statements in the loop nest. Dynamic statement instances are mapped to the product space by means of *embedding functions.* These functions are affine expressions of the loop indices that surround the statement and symbolic constants present in the code. They generalize such code transformations like code-sinking, fusion and fission.

As discussed in Chapter 3, these embedding functions can be chosen so that the resulting product space has certain desirable qualities. Section 3.1 describes how to choose embedding functions that create a fully permutable product space. Such a space can be used in order to apply further transformations like interchange and tiling in order to improve locality. Section 3.1.2 discusses how to choose the right embeddings that not only allow tiling and permutation of the product space but also enhance locality by reducing the reuse distance between statement instances that exhibit reuse. Chapter 4 shows how the product space formulation also enables us to generate block-recursive versions of the codes which are portable and thus can be used for writing libraries.

A prototype of the above technology has been implemented. Many important kernels in numerical applications like Cholesky, Jacobi, Red-Black Gauss Seidel, Triangular solve etc. show substantial performance improvement as a result of applying this technology. The technology has also been shown to apply to substantially larger codes like the Tomcatv benchmark from SpecFP95 suite.

5.2 Future Work

As this dissertation has demonstrated, the product space is a powerful formulation that allows the transformation of imperfectly-nested loop nests for locality in a single unified framework. There are many interesting questions that are open at the end of this dissertation. This section briefly describes the possible avenues for future research.

1. Application to large programs : A program can itself be considered to be a huge imperfectly nested loop nest. So the technology developed in this thesis can be applied to the entire program. A major concern in doing this is the scalability of this framework. For large codes, the product space could potentially be very huge and intractable. This presents two problems –

   (a) The algorithm to determine embeddings a discussed in Section 3.3.4 is in the worst case quadratic in the number of dimensions of the product space.

   (b) Fourier-Motzkin elimination which is at the heart of applying Farkas's lemma is theoretically exponential in the number of variables and constraints.

In practice, for small codes, the algorithms can be implemented efficiently. But, this may change when applied to larger problems.

There are many ways the above issue can be tackled. Firstly, in a setting such as in a production compiler, the technology can be applied to smaller segments of the code. In this case, the compiler must decide how the code is to be segmented. Secondly, it may not be necessary to use the entire product space. Heuristics can be used to limit the number of choices for an embedding function.

2. When dependence analysis fails : Recent research such as Fractal Symbolic Analysis [25] has demonstrated that dependence analysis is too strict to allow transformations on certain applications even when the transformations are legal. For example, in kernels like LU factorization with partial pivoting, there exists a legal way of tiling the loops. But any technology that solely depends on dependence analysis cannot show the legality of this transformation because certain dependences are violated. Symbolic analysis is required to show that these dependences can in fact be violated since the code after restructuring is semantically equivalent to the original code.

   It is an interesting problem to see how the results of Fractal Symbolic Analysis can be summarized and used within the framework of the product space in order to be able to synthesize the tiling transformation for codes such as LU with partial pivoting.

3. Block size determination : In this thesis, we have described a straight-forward algorithm that determines tile-sizes for the tiled dimensions of the transformed product space. This algorithm ignores the effects of conflict misses which could

make a huge difference to the performance. Additional research is required to study these effects and develop an algorithm that can reduce or eliminate cache conflicts.

4. Recursive code generation : An important parameter in the generation of block-recursive versions of a code is the size of the base-block when the control structure changes from recursive to iterative. As discussed in Section 4.4.1, the overhead due to recursion can be substantial for small base sizes. The base size should thus be chosen so that

   (a) the overhead due to recursion is small

   (b) the compiler backend can schedule the base blocks effectively

   This thesis does not address the issue of base block size selection, merely observing that a block size of 32 seems to work well for the two examples discussed. This base size can either be determined empirically or a cost model can be developed to choose an appropriate base-block size.

   Another issue that has not been resolved by this thesis is the effect of block-recursive codes on TLB misses. Experiments discussed in Section 4.4.2 show that block-recursive version of codes interact badly with the TLB for large array sizes. Research is required to further study this issue. One possibility of counteracting this effect is to store the array in recursive data formats like space-filling or block-recursive formats [7].

5. Data transformations : This thesis does not address the issue of also transforming the data in addition restructuring the program. Previous research [8], have shown that certain programs can benefit from an approach that combines data transformations and loop transformations.

Bibliography

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Principle and Practice of Parallel Programming*, pages 39–50, Apr. 1991.

[2] 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.

[3] E. Ayguadé and J. Torres. Partitioning the statement per iteration space using nonsingular matrices. In *1993 ACM International Conference on Supercomputing*, pages 407–415, Tokyo, July 1993.

[4] U. Banerjee. A theory of loop permutations. In *Languages and compilers for parallel computing*, pages 54–74, 1989.

[5] U. Banerjee. Unimodular transformations of double loops. In *Languages and compilers for parallel computing*, pages 192–219, 1990.

[6] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. *ACM SIGPLAN Notices*, 29(11):252–262, Nov. 1994.

[7] S. Chaterjee, V. Jain, A. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *International Conference on Supercomputing (ICS'99)*, June 1999.

[8] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared memory machines. In *SIGPLAN 1995 conference on Programming Languages Design and Implementation*, June 1995.

[9] P. Claus. Counting solutions to linear and nonlinear constraints through Erhart polynomials. In *"ACM International Conference on Supercomputing.* ACM, May 1996.

[10] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI).* ACM Press, June 1995.

[11] P. Feautrier. Some efficient solutions to the affine scheduling problem - part 1: one dimensional time. *International Journal of Parallel Programming*, Oct. 1992.

[12] P. Feautrier. Some efficient solutions to the affine scheduling problem - part ii: multi-dimensional time. *International Journal of Parallel Programming*, Dec. 1992.

[13] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 11th International Conference on Supercomputing (ICS-97)*, pages 317–324, New York, July7–11 1997. ACM Press.

[14] G. Golub and C. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[15] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, Nov. 1997.

[16] W. Kelly and W. Pugh. Selecting affine mappings based on performance estimation. *Parallel Processing Letters*, 4(3):205–209, Sept. 1994.

[17] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *5th Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, Feb. 1995.

[18] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *1992 ACM International Conference on Supercomputing*, pages 323–334, Washington, D.C., July 1992. ACM Press.

[19] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Programming Languages, Design and Implementation*. ACM SIGPLAN, June 1997.

[20] S. Kung. *VLSI Array Processors*. Prentice-Hall Inc, 1988.

[21] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Apr. 8–11, 1991.

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

[23] W. Li and K. Pingali. A singular loop transformation based on non-singular matrices. *International Journal of Parallel Programming*, 22(2), Apr. 1994.

[24] A. Lim and M. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24:445–475, 1998.

[25] V. Menon. *Symbolic Computation Techniques for Array Computations*. PhD thesis, Cornell University, Computer Science, Aug. 2000.

[26] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Communications of the ACM*, pages 102–114, Aug. 1992.

[27] W. Pugh. Counting solutions to presburger formulas: How and why. Technical report, University of Maryland, 1993.

[28] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, Oct. 1992.

[29] J. M. Ramesh C. Agarwal, Fred G. Gustavson and S. Schmidt. Engineering and Scientific Subroutine Library Release 3 for IBM ES/3090 Vector Multiprocessors. *IBM Systems Journal*, 28(2):345–350, 1989.

[30] V. 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.

[31] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *SIGPLAN99 conference on Programming Languages, Design and Implementation*, June 1999.

[32] M. Wolf and M. Lam. A data locality optimizing algorithm. In *SIGPLAN 1991 conference on Programming Languages Design and Implementation*, June 1991.

[33] M. E. Wolf and M. S. Lam. An algorithmic approach to compound loop transformations. In *Languages and compilers for parallel computing*, pages 243–273, 1990.

[34] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *MICRO 29*, pages 274–286, Silicon Graphics, Mountain View, CA, 1996.

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

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