Synthesizing Transformations for Locality Enhancement of Imperfectly-nested Loop Nests

Nawaaz Ahmed, Nikolay Mateev, Keshav Pingali

Department of Computer Science

Cornell University

Ithaca, NY 14853

September 24, 2001

[1]

Abstract

[1]This work was supported by NSF grants EIA-9726388, ACI-9870687, EIA-9972853, and ACI-0085969.
Corresponding author: Keshav Pingali, Department of Computer Science, Cornell University, Ithaca, NY 14853; email:pingali@cs.cornell.edu, tel: (607)-255-7203, fax: (607)-255-4428.

Linear loop transformations and tiling are known to be very effective for enhancing locality of reference in perfectly-nested loops. However, they cannot be applied directly to imperfectly-nested loops. Some compilers attempt to convert imperfectly-nested loops into perfectly-nested loops by using statement sinking, loop fusion, etc., and then apply locality enhancing transformations to the resulting perfectly-nested loops, but the approaches used are fairly ad hoc and may fail even for simple programs.

In this paper, we present a systematic approach for synthesizing transformations to enhance locality in imperfectly-nested loops. The key idea is to embed the iteration space of each statement into a special iteration space called the *product space*. The product space can be viewed as a perfectly-nested loop nest, so embedding generalizes techniques like statement sinking and loop fusion which 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, our embeddings are chosen carefully to enhance locality. The product space can itself be transformed to increase locality further, after which fully permutable loops can be tiled. The final code generation step may produce imperfectly-nested loops as output if that is desirable.

We present experimental evidence for the effectiveness of this approach,

using dense numerical linear algebra benchmarks, relaxation codes, and the tomcatv code from the SPEC benchmarks.

Keywords: Automatic locality enhancement, restructuring compilers, caches, program transformation, imperfectly-nested loops

1 Background and Previous Work

Sophisticated program transformation algorithms based on polyhedral algebra have been developed for enhancing locality of reference in perfectly-nested loops[2]. Highlights of this technology are the following. The iterations of the loop nest are modeled as points in an integer lattice, and linear loop transformations are modeled as nonsingular integer matrices mapping one lattice to another. A sequence of linear loop transformations is modeled by the product of the matrices representing the individual transformations; since the set of nonsingular matrices is closed under matrix product, a sequence of linear loop transformations can itself be represented by a nonsingular matrix. The problem of finding an optimal sequence of linear loop transformations to enhance locality of reference is thus reduced to the problem of finding an integer matrix that satisfies some desired property, permit-

[2]A set of loops is said to be perfectly-nested if all assignment statements in the loop nest are contained in the innermost loop of that loop nest.

ting the full machinery of matrix methods and lattice theory to be applied to this problem.[3–5,22,29,31]

This technology is fairly mature, and it has been incorporated into production compilers, enabling these compilers to produce good code for perfectly-nested loop nests. However, most programs contain *imperfectly-nested* loop nests in which assignment statements are contained in some but not all of the loops of the loop nest. For example, important matrix factorizations like Cholesky, LU and QR factorizations[12] are all imperfectly-nested loop nests. An entire procedure, which is usually a sequence of perfectly- or imperfectly-nested loop nests, can itself be considered to be an imperfectly-nested loop nest.

[Figure 1 about here.]

As an example, consider the Jacobi code fragment in Figure 1 which is typical of programs that solve partial differential equations (pde's) by explicit methods. These *relaxation codes* 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 of the pde. In the Jacobi code, statements `S1` and `S2` touch the same data in each iteration of the `t` loop; furthermore, instances of these statements in different iterations of the time loop touch the same

data as well. This data reuse will not be exploited if the `L` and `A` arrays do not fit into the cache.

Data reuse between instances of `S1` and `S2` in a given iteration of the `t` loop can be improved by rewriting the Jacobi code as shown in Figure 2. This code can be obtained by peeling the first iterations of the `i1` and `j1` loops, and the last iterations of the `i2` and `j2` loops, and then fusing the remaining `i1` and `i2` loops, and the `j1` and `j2` loops. Exploiting data reuse between different iterations of the time loop requires even more elaborate transformations: the `i` and `j` loops must be skewed by `2*t` and all loops must be tiled; finally, if the arrays are stored in column major order, the `i` and `j` loops must be interchanged to permit exploitation of spatial locality as well. One version of this final code is shown in Figure 25. This code is obviously much more complex than the code shown in Figure 1, so it is desirable to have compiler technology that can improve performance automatically by restructuring.

[Figure 2 about here.]

A number of such restructuring strategies have been proposed for enhancing locality in imperfectly-nested loop nests like matrix factorization codes and Jacobi.

Special-purpose techniques for tiling matrix factorization codes have been proposed by Carr and Kennedy.[6] They studied hand-blocked codes in the LAPACK library, and identified sequences of loop transformations that could be used to transform point codes into the corresponding block codes. These sequences are relevant only to factorization codes, and are not useful for locality enhancement of relaxation codes for example. Special-purpose transformations for locality enhancement in relaxation codes were proposed recently by Song and Li,[30] but these transformations cannot be used for locality enhancement of matrix factorization codes.

General-purpose techniques for locality enhancement are obviously preferable to special-purpose techniques. A simple and general approach is to restructure separately each perfectly-nested loop in an imperfectly-nested loop nest, using perfectly-nested loop technology. However, the performance improvement obtained with this strategy can be quite limited.[18] For example, this strategy obviously fails to generate the desired transformation for the Jacobi code discussed above. Another approach used by current commercial compilers 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*,[16,33] and then (ii) use locality enhancement techniques for the resulting maximal perfectly-

nested loops. For most programs, 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 these transformations might lead to code that cannot be tiled, while other orders could result in tilable code.(17) A further complication is that loop fission and fusion are themselves useful in improving data locality of loop nests; for example, loop fusion improves inter-loop-nest reuse in the Jacobi example.

In this paper, we describe a general approach to locality enhancement of imperfectly-nested loops which extends the standard approach used for perfectly-nested loops. A pictorial representation of our approach is shown in Figure 3. The iteration space of each statement is embedded in a Cartesian space called the *product space*, using affine embedding functions F_i. These embeddings are chosen so as to improve reuse in the program, and they generalize transformations like loop fusion and loop fission that have been used in the literature for locality enhancement. The product space itself can be viewed as a perfectly-nested loop nest (although one which has many redundant dimensions as we discuss later in this paper), so embeddings can also be viewed as a generalization of techniques currently used to transform imperfectly-nested loops into perfectly-nested ones. The product space is further transformed using approaches developed for perfectly-

nested loops, such as height reduction;[21] when possible, loops are made fully permutable, enabling them to be tiled. These transformations are represented in Figure 3 by the non-singular matrix T. After projecting out redundant dimensions, code is generated using standard techniques from polyhedral algebra; this code generation process may produce imperfectly-nested loop nests if appropriate.

The rest of this paper is organized as follows. Section 2 describes an abstract framework for locality enhancement of imperfectly-nested loop nests. Section 3 describes the concrete approach to locality enhancement we use in this paper. Because the complete algorithm is complicated, we work through an example in Section 4 to highlight important aspects of our approach. Section 5 gives the details of the algorithm. Section 6 evaluates the effectiveness of this algorithm in enhancing locality of programs on the SGI Octane workstations. We show that our approach automatically performs the desired transformations on the Jacobi code fragment discussed above. Finally, Section 7 compares our approach with other approaches in the literature.

[Figure 3 about here.]

2 An Abstract Model of Locality Enhancement

[Figure 4 about here.]

This section provides an abstract view of how program transformations and locality enhancement are modeled by polyhedral methods.

2.1 Program Execution Model

A program is assumed to consist of statements contained in some number of loop nests. All loop bounds and array access functions are assumed to be affine functions of surrounding loop indices. We will use $\mathtt{S}_1$, $\mathtt{S}_2$, ..., $\mathtt{S}_n$ to name the statements in the program in syntactic order. A *dynamic instance* of a statement $\mathtt{S}_k$ refers to a particular execution of the statement for a given value of index variables $\vec{\imath}_k$ of the loops surrounding it, and is represented by $\mathtt{S}_k(\vec{\imath}_k)$.

Program execution induces a total order on the dynamic instances of a statement $\mathtt{S}_k$. This *statement execution order* is modeled by the *statement iteration space*, denoted by $\mathcal{S}_k$, as follows.

Definition 1 *The* statement iteration space $\mathcal{S}_k$ *of a statement* $\mathtt{S}_k$ *is a Cartesian space defined as follows.*

1. *Construct a Cartesian space $\mathcal{S}_k$ of dimension equal to the number of loops surrounding* $\mathtt{S}_k$*.*

2. *Map dynamic instances of* $\mathtt{S}_k$ *to* $\mathcal{S}_k$ *so that the following conditions are satisfied:*

   (a) *At most one statement instance is mapped to a point in the space* $\mathcal{S}_k$*.*

   (b) *If the points in space* $\mathcal{S}_k$ *are traversed in lexicographic order, and any statement instance mapped to a point is executed when that point is visited, the statement execution order is reproduced.*

Similarly, program execution induces a total order on the dynamic instances of *all* statements. We will call this order the *original execution order* of the program, and model it formally by a Cartesian space $\mathcal{P}$, called the *program iteration space*, and embeddings functions $\tilde{\mathcal{F}} = \{\tilde{F}_1, \tilde{F}_2, \ldots, \tilde{F}_n\})$ where $\tilde{F}_k$ maps statement iteration space $\mathcal{S}_k$ into $\mathcal{P}$.

Definition 2 *The* original execution order *of a program is modeled as a pair* $(\mathcal{P}, \tilde{\mathcal{F}} = \{\tilde{F}_1, \tilde{F}_2, \ldots, \tilde{F}_n\})$ *satisfying the following conditions.*

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

2. *The embedding function $\tilde{F}_k$ maps statement iteration space $\mathcal{S}_k$ into $\mathcal{P}$, and satisfies the following constraints.*

   (a) *$\tilde{F}_k$ is one-to-one*[3].

   (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.*

For example, the original execution order of the code fragment shown in Figure 4(a) can be represented by mapping statement instances to the two-dimensional space shown in Figure 4(b). In general, there are many models for the original execution order of a program. Figure 4(c) shows a one-dimensional Cartesian space and associated embeddings which model the original execution order of the program of Figure 4(a).

It is not obvious that a pair $(\mathcal{P}, \tilde{F})$ can always be found to model the program execution order since it is not obvious that appropriate embedding functions can be found. In Section 3.2.1, we show how a "canonical" pair can always be constructed for any program.

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

2.2 Program Transformation

Execution orders different from the original execution order can be represented by appropriate pairs $(\mathcal{P}, \mathcal{F})$. We optimize programs therefore by transforming the original execution order $(\mathcal{P}, \tilde{\mathcal{F}})$ to another execution order $(\mathcal{P}, \mathcal{F})$ which (i) preserves the semantics of the program, and (ii) is better for locality.

To understand the relationship between embeddings and transformations, it is instructive to examine how code can be generated for a given space $\mathcal{P}$ and embeddings $\mathcal{F}_i$. A simple approach is to traverse the points in the space $\mathcal{P}$ in lexicographic order, and execute all statement instances that get mapped to a given point of $\mathcal{P}$ when that point is visited (if multiple statement instances get mapped to a given point, those instances are executed in the original program order). For example, consider the space and embeddings shown in Figure 5(a,b) for the program of Figure 4(a). Naive code for the transformed program is shown in Figure 5(c). This code can be optimized using standard polyhedral techniques to produce the code shown in Figure 5(d).

By choosing the space and embeddings shown in Figure 5(a,b), we have in effect chosen to jam the two loops in the source code. A similar effect can be obtained for a 1-D Cartesian space by choosing the embeddings shown in Figure 6.

[Figure 5 about here.]

[Figure 6 about here.]

2.3 Legality of Transformations

The original execution order $(\mathcal{P}, \tilde{\mathcal{F}})$ can be legally transformed to another execution order $(\mathcal{P}, \mathcal{F})$ if the latter preserves the semantics of the program. The semantics of a program will be preserved if all dependences are preserved by the transformation.

Formally, a dependence exists from instance $\vec{\imath}_s$ of statement $\mathtt{S}_s$ (the source of the reuse) to instance $\vec{\imath}_d$ of statement $\mathtt{S}_d$ (the destination) if the following conditions are satisfied:

1. *Loop bounds*: Both the 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 $B_s * \vec{\imath}_s + b_s \geq 0$ and $B_d * \vec{\imath}_d + b_d \geq 0$ 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 $A_s * \vec{\imath}_s + a_s$ and $A_d * \vec{\imath}_d + a_d$. Hence the existence

of a reuse requires that $A_s * \vec{\imath}_s + a_s = A_d * \vec{\imath}_d + a_d$.

3. *Precedence order*: Instance $\vec{\imath}_s$ of statement $\mathtt{S}_s$ occurs before instance $\vec{\imath}_d$ of statement $\mathtt{S}_d$ in program execution order. If $common_{sd}$ is a function that returns the loop index variables of the loops common to both $\vec{\imath}_s$ and $\vec{\imath}_d$, this condition can be written as $common_{sd}(\vec{\imath}_d) \succeq common_{sd}(\vec{\imath}_s)$ if $\mathtt{S}_d$ follows $\mathtt{S}_s$ syntactically or $common_{sd}(\vec{\imath}_d) \succ common_{sd}(\vec{\imath}_d)$ 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 $X_s * \vec{\imath}_s - X_d * \vec{\imath}_d + x \geq 0$.

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.

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

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

represents a polyhedron.

For the example in Figure 4(a), the flow dependence from $\mathtt{S1}(i_1)$ to $\mathtt{S2}(i_2)$ can be represented by the dependence class $\mathcal{D}_1 = \{(i_1, i_2) : 1 \leq i_1, i_2 \leq N, i_1 = i_2\}$. It is straightforward to represent these inequalities as matrix inequalities, but we will not do so to keep the discussion simple.

Let $\mathcal{F} = \{F_1, F_2, \dots F_n\}$ be a set of embedding functions for a program. We will say that these embedding functions are *legal* if for every $(\vec{\imath}_s, \vec{\imath}_d)$ in a dependence class, the point that $\vec{\imath}_s$ is mapped to in the program iteration space is lexicographically less than the point that $\vec{\imath}_d$ is mapped to. For future reference, we define this formally.

Definition 3 *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, the following condition is true:*

$$\forall (\vec{\imath}_s, \vec{\imath}_d) \in \mathcal{D} \;\; F_d(\vec{\imath}_d) \succeq F_s(\vec{\imath}_s)$$

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

2.4 Transformations to Promote Reuse

In the program of Figure 4(a), 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 called the *reuse distance*. In the example, the reuse distance is $N - 1$. If we represent the execution order for the example 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 at most a constant number of statement instances (one from each statement) mapped to each point in the space.

Reducing reuse distance increases the likelihood that the data will be in the cache when the second statement instance tries to access it. For our example, we can reduce the reuse distances from $N-1$ to zero by using the embeddings shown in Figure 5. As discussed before, these embeddings correspond to jamming the two loops of Figure 4(a), which is obviously good for enhancing data reuse.

2.4.1 Reuse Classes

The formal definition of reuse is similar to that of dependence in Section 2.3. A reuse exists from instance $\vec{\imath}_s$ of statement $\texttt{S}_s$ (the source of the reuse) to instance $\vec{\imath}_d$ of statement $\texttt{S}_d$ (the destination) if the following conditions are satisfied:

1. *Loop bounds*: Both the 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 $B_s * \vec{\imath}_s + b_s \geq 0$ and $B_d * \vec{\imath}_d + b_d \geq 0$ 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 $A_s * \vec{\imath}_s + a_s$ and $A_d * \vec{\imath}_d + a_d$. Hence the existence of a reuse requires that $A_s * \vec{\imath}_s + a_s = A_d * \vec{\imath}_d + a_d$.

3. *Precedence order*: Instance $\vec{\imath}_s$ of statement $\texttt{S}_s$ occurs before instance $\vec{\imath}_d$ of statement $\texttt{S}_d$ in program execution order. If $common_{sd}$ is a function that returns the loop index variables of the loops common to both $\vec{\imath}_s$ and $\vec{\imath}_d$, this condition can be written as $common_{sd}(\vec{\imath}_d) \succeq common_{sd}(\vec{\imath}_s)$ if $\texttt{S}_d$ follows $\texttt{S}_s$ syntactically or $common_{sd}(\vec{\imath}_d) \succ common_{sd}(\vec{\imath}_d)$ 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 $X_s * \vec{i}_s - X_d * \vec{i}_d + x \geq 0$.

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.

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

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

For the example in Figure 4(a), the temporal reuse between $\texttt{S1}(i_1)$ and $\texttt{S2}(i_2)$ can be represented by the reuse class $\mathcal{R}_1 = \{(i_1, i_2) : 1 \leq i_1, i_2 \leq N, i_1 = i_2\}$ which can be represented by matrix inequalities in the obvious way.

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 $A_d * \vec{\imath}_d + a_d - A_s * \vec{\imath}_s - a_s$ 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 4(a), there is spatial reuse between instances of the two statements. For a cache line containing 4 array elements, the spatial reuse can be represented by the reuse class $\mathcal{R}_2 = \{(i_1, i_2) : 1 \leq i_1, i_2 \leq N, 1 \leq i_2 - i_1 \leq 3\}$.

2.4.2 Promoting Data Reuse

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 $(\vec{\imath}_s, \vec{\imath}_d) \in \mathcal{R}$. Let $Distance(\vec{\imath}_s, \vec{\imath}_d)$ be the number of points in the space $\mathcal{P}$ with statements mapped to them that lie lexicograph-

[4] For Fortran storage order.

ically between $F_s(\vec{\imath}_s)$ and $F_d(\vec{\imath}_d)$. The reuse distance between $\mathtt{S}_s(\vec{\imath}_s)$ and $\mathtt{S}_d(\vec{\imath}_d)$ is proportional to $Distance(\vec{\imath}_s, \vec{\imath}_d)$. Given some ordering of the reuse pairs in a program, we can define a vector $Distances(\mathcal{P}, \mathcal{F})$ where $Distances(\mathcal{P}, \mathcal{F})(k)$ is the value of $Distance(\vec{\imath}_s, \vec{\imath}_d)$ under the execution order $(\mathcal{P}, \mathcal{F})$ for the kth reuse pair $(\vec{\imath}_s, \vec{\imath}_d)$.

One formulation of locality enhancement is to find legal embeddings $\mathcal{F}_{opt}$ that minimize $\|Distances(\mathcal{P}, \mathcal{F})\|_X$ for a suitable norm $\|\cdot\|_X$.

We develop this theme next.

3 A Concrete Model of Locality Enhancement

We now develop a practical algorithm for locality enhancement based on the abstract model in Section 2. We restrict embedding functions to be affine, and define a special space called the *product space* which we argue is the right space for locality enhancement considerations. We then present our approach to minimizing reuse distances in the product space.

3.1 Product Spaces and Embedding Functions

Definition 4 *The* product space $\mathcal{P}$ *for a program is the Cartesian product of all the statement iteration spaces of the statements in that program. The order in which this product is formed is the syntactic order in which the statements appear in the program. The dimension of the product space is denoted by* $|\mathcal{P}|$.

For the Jacobi code in Figure 1, the iteration space $\mathcal{S}_1$ of statement `S1` is the three-dimensional space $t_1 \times i_1 \times j_1$, while the iteration space $\mathcal{S}_2$ of `S2` is the three-dimensional space $t_2 \times i_2 \times j_2$. The product space is the Cartesian product of these two spaces and hence is the six dimensional space $t_1 \times i_1 \times j_1 \times t_2 \times i_2 \times j_2$.

The relationship between statement iteration spaces and the product space is specified by projection and embedding functions. Suppose $\mathcal{P} = \mathcal{S}_1 \times \mathcal{S}_2 \times \ldots \times \mathcal{S}_n$. We restrict our embedding functions to be *affine* to permit the use of integer linear programming techniques. 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 the Jacobi code in Figure 1,

$$\pi_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} I_{3\times3} & 0 \end{bmatrix},$$

$$\pi_2 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & I_{3\times3} \end{bmatrix}.$$

The embedding functions we allow in our framework can be defined conveniently in terms of these projections.

Definition 5 Embedding function F_i *maps statement iteration space* $\mathcal{S}_i$ *to the product space* $\mathcal{P}$. *We consider only those embedding functions* $F_i : \mathcal{S}_i \rightarrow \mathcal{P}$ *that satisfy the condition* $\pi_i(F_i(q)) = q$ *for all* $q \in \mathcal{S}_i$.

Intuitively, this condition requires that the statement iteration space of a statement `S` be mapped to itself in the subspace of the product space corresponding to that statement. This restriction keeps the development simple. Each F_i is therefore automatically one-to-one, but points from two different statement iteration spaces may be mapped to a single point in the product space.

3.2 Properties of Embeddings

Pairs $(\mathcal{P}, \mathcal{F})$ as restricted by Definition 5 can represent only a restricted set of execution orders, so it is reasonable to wonder if our framework is general enough to model the transformations required for locality enhancement of imperfectly-nested loops. In this section, we argue that this is indeed the case by making two points. In Section 3.2.1, we show that we can always find embeddings that model the original execution order of any program, so at the very least, we can model the execution of the original program. In Section 3.2.2, we show that transformations like statement sinking, loop fission, loop fusion, and skewed loop fusion which are important for locality enhancement of imperfectly-nested loops can be modeled in this framework.

3.2.1 Embeddings for original program order

[Figure 7 about here.]

Consider the perfectly-nested loop code in Figure 7. It is easy to verify that this code is equivalent to the Jacobi source program shown in Figure 1 in the sense that the execution orders of statement instances in both programs are identical. Intuitively, the loops in Figure 7 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:

$$F_1(\begin{bmatrix} t_1 \\ i_1 \\ j_1 \end{bmatrix}) = \begin{bmatrix} t_1 \\ i_1 \\ j_1 \\ t_1 \\ 2 \\ 2 \end{bmatrix} \quad F_2(\begin{bmatrix} t_2 \\ i_2 \\ j_2 \end{bmatrix}) = \begin{bmatrix} t_2 \\ N-1 \\ N-1 \\ t_2 \\ i_2 \\ j_2 \end{bmatrix}.$$

To understand how these embeddings were obtained, consider F_1.

1. Definition 5 requires that each statement iteration space be mapped to itself in the sub-space corresponding to that statement in the product space. Therefore, F_1 must be of the following form:

$$F_1(\begin{bmatrix} t_1 \\ i_1 \\ j_1 \end{bmatrix}) = \begin{bmatrix} t_1 \\ i_1 \\ j_1 \\ .... \end{bmatrix}$$

2. In the sub-space corresponding to $\mathtt{S}_2$ in the product space, dimension t_2 arises from the t loop which is common to both $\mathtt{S}_1$ and $\mathtt{S}_2$. The embedding into dimension t_2 is chosen to the same as the embedding into dimension t_1 of the product space. Therefore, F_1 is of the following form:

$$F_1(\begin{bmatrix} t_1 \\ i_1 \\ j_1 \end{bmatrix}) = \begin{bmatrix} t_1 \\ i_1 \\ j_1 \\ t_1 \\ .... \end{bmatrix}$$

3. Finally, consider dimensions i_2 and j_2 of the product space that arise from loops that do not surround statement $\mathcal{S}_1$. Since statement S_1 is syntactically before statement S_2, we choose the smallest values of the indices of the i_2 and j_2 loops as the value of F_1 for these co-ordinates; had S_1 occurred syntactically after statement S_2, we would have chosen the largest value of the indices of these loops.

   Therefore, F_1 is

$$F_1(\begin{bmatrix} t_1 \\ i_1 \\ j_1 \end{bmatrix}) = \begin{bmatrix} t_1 \\ i_1 \\ j_1 \\ t_1 \\ 2 \\ 2 \end{bmatrix}$$

In general, we can embed any code into its product space as follows. Let $F_k : \mathcal{S}_k \rightarrow \mathcal{P}$ be an 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 $\mathtt{S}_l$ are denoted by $F_{k,l}$. Our initial requirement on embedding functions can be summarized by $F_{k,k}(\vec{\imath}_k) = \vec{\imath}_k$.

$F_{k,l}$ maps statement S_k to the dimensions of the product space corresponding to statement S_l. These dimensions can be separated in two groups—dimensions corresponding to loops common to S_k and S_l in the original code, and dimensions corresponding to loops surrounding S_l but not S_k.

Definition 6 *Consider two statements S_k and S_l. We divide the product space dimensions corresponding to S_l in two groups:*

1. *$C_{k,l}$ are the dimensions corresponding to loops in $common_{kl}$, where $common_{kl}$ is defined as in Section 2.3 to be a function that returns the loop index variables of the loops common to both $\vec{\imath}_k$ and $\vec{\imath}_l$;*
2. *$N_{k,l}$ are the dimensions corresponding to loops surrounding S_l but not S_k.*

For the Jacobi example, $common_{12} = t$, hence $C_{1,2} = (t_2)$, and $N_{1,2} = (i_2, j_2)^T$. Similarly, $C_{2,1} = (t_1)$, and $N_{2,1} = (i_1, j_1)^T$.

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, $\min_{\prec}(i_2, j_2)^T = (2,2)^T$ and $\max_{\prec}(i_i, j_1)^T = (N-1, N-1)^T$.

Definition 7 Canonical Embeddings

1. $F_{k,k}(\vec{\imath}_k) = \vec{\imath}_k$;
2. $F^j_{k,l}(\vec{\imath}_k) = \vec{\imath}^j_k$ *for dimensions* $j \in C_{k,l}$;
3. $F^j_{k,l}(\vec{\imath}_k) = (\min_{\prec}(N_{k,l}))^j$ *if* $l > k$*, or* $F^j_{k,l}(\vec{\imath}_k) = (\max_{\prec}(N_{k,l}))^j$ *if* $l < k$*, for dimensions* $j \in N_{k,l}$.

Theorem 8 *The canonical embeddings in Definition 7 represent the original execution order.*

Proof: Consider two statement instances, $S_k(\vec{\imath}_k)$ and $S_l(\vec{\imath}_l)$. Suppose that $\vec{\imath}_k$ occurs before $\vec{\imath}_l$ in the original program execution order. That means that $common_{kl}(\vec{\imath}_l) \succeq common_{kl}(\vec{\imath}_k)$ if S_l follows S_k syntactically (i.e. $l > k$), or $common_{kl}(\vec{\imath}_l) \succ common_{kl}(\vec{\imath}_k)$ if it does not (i.e. $l < k$).

For a set of embeddings to preserve the original program order, we must show that $F_l(\vec{\imath}_l) - F_k(\vec{\imath}_k)$ is lexicographically non-negative. (If $F_l(\vec{\imath}_l) - F_k(\vec{\imath}_k) = \vec{0}$, then the instances are mapped to the same point and will be executed in original program order per Definition 2.)

We prove a stronger statement: that for any m, $1 \leq m \leq n$, $F_{l,m}(\vec{i}_l) - F_{k,m}(\vec{i}_k)$ is lexicographically positive if $l < k$, or lexicographically non-negative if $l > k$.

First consider the case $m = k$. For dimensions $j \in C_{k,l}$, $F^j_{l,k}(\vec{i}_l) - F^j_{k,k}(\vec{i}_k) = common^j_{kl}(\vec{i}_l) - common^j_{kl}(\vec{i}_k)$ by our definition, and is therefore non-negative. If $l < k$, it is strictly positive, and our claim holds. If $l > k$, it may be zero and we need to consider the remaining dimensions of $F_l(\vec{i}_l) - F_k(\vec{i}_k)$. For those dimensions $j \in N_{k,l}$, $F^j_{l,k}(\vec{i}_l) - F^j_{k,k}(\vec{i}_k) = (\max_{\prec}(N_{l,k}))^j - \vec{i}^j_k$ because of parts (1) and (3) of Definition 7. That vector is lexicographically non-negative, so our claim holds.

The argument for the case $m = l$ is almost identical and is omitted.

The interesting case is $m \neq k, l$. Assume that $m < k < l$. (The argument for other orders is similar and is omitted.) There are three subcases:

1. $C_{k,m} = C_{l,m}$: In that case, $common_{km} = common_{lm} \subseteq common_{kl}$ and $F^j_{l,m}(\vec{i}_l) - F^j_{k,m}(\vec{i}_k) = \vec{i}^j_l - \vec{i}^j_k$ for all dimensions $j \in C_{k,m}$. For the remaining dimensions $j \in N_{k,m}$, $F^j_{l,m}(\vec{i}_l) - F^j_{k,m}(\vec{i}_k) = 0$ by Definition 7(3). Our claim clearly holds.

2. $C_{k,m} \subset C_{l,m}$: In that case, $common_{km} = common_{kl} \subset common_{lm}$. That is not possible under our assumption that $m < k < l$.

3. $C_{l,m} \subset C_{k,m}$: In that case, $common_{lm} = common_{kl} \subset common_{km}$. $F^j_{l,k}(\vec{i}_l) - F^j_{k,k}(\vec{i}_k) = \vec{i}^j_l - \vec{i}^j_k$ for all dimensions $j \in C_{l,m}$. For dimensions $j \in N_{k,m}$, $F^j_{l,m}(\vec{i}_l) - F^j_{k,m}(\vec{i}_k) = 0$ by Definition 7(3). For dimensions $j \in N_{l,m} \cap C_{k,m}$,

$F^j_{l,m}(\vec{i_l}) - F^j_{k,m}(\vec{i_k}) = (\max_{\prec}(N_{l,m}))^j - \vec{i}^j_k$. As all three components of the vector are non-negative, our claim holds.

Therefore, $F_l(\vec{i_l}) - F_k(\vec{i_k})$ is lexicographically non-negative, and the canonical embeddings result in the original execution order. □

3.2.2 Expressive Power of Embeddings

The product space and affine embeddings framework cannot be used to model the following loop transformations.

1. Index-set Splitting: Since all the instances of a particular statement are mapped with the same affine embedding function, explicit index-set-splitting is not 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.

It should be noted that our code generation step may perform index-set splitting and tiling after other transformations are performed, as discussed in Section 5.2.

On the other hand, the product space and affine embeddings framework is sufficient to capture most common loop transformations such as code-sinking,

loop-fission and loop-fusion which are used in current compilers such as the SGI MIPSPro to convert imperfectly-nested loop nests into perfectly-nested ones.

Rather than prove a formal result, we illustrate loop fission, fusion, and statement sinking using the Jacobi example. Note that these transformations are not necessarily legal for the Jacobi code.

Figure 8 illustrates loop fission of the Jacobi code shown in Figure 1. After loop fission, all instances of statement `S1` in Figure 1 are executed before all instances of statement `S2`. The resulting code is shown in Figure 8(a), and the embedding functions for the two statements are shown in Figure 8(b). Note that this execution order is not legal for the original program.

[Figure 8 about here.]

[Figure 9 about here.]

Loop fusion was discussed in Figure 5. Figure 9 illustrates fusion for the Jacobi example. In this code, the two pairs of `i` and `j` loops in Figure 1 are fused together. The corresponding embeddings are shown in Figure 9)(b); they map statement instances $S_1(t, i, j)$ and $S_2(t, i, j)$ to the same point (t, i, j, t, i, j) in the product space. Note that fusing the two loop nests in the original code is not legal—to fuse them legally, the two loop-nests must be skewed with respect to

each other before fusion. Such a transformation is known as *skewed fusion* and produces the optimized Jacobi version of Figure 2 which is legal. The corresponding embeddings are the following:

$$F_1(\begin{bmatrix} t_1 \\ i_1 \\ j_1 \end{bmatrix}) = \begin{bmatrix} t_1 \\ i_1 \\ j_1 \\ t_1 \\ i_1 - 1 \\ j_1 - 1 \end{bmatrix} \quad F_2(\begin{bmatrix} t_2 \\ i_2 \\ j_2 \end{bmatrix}) = \begin{bmatrix} t_2 \\ i_2 + 1 \\ j_2 + 1 \\ t_2 \\ i_2 \\ j_2 \end{bmatrix}$$

Finally, the transformed code corresponding to the original execution order shown in Figure 7 is an example of a generalized version of code-sinking. Here, instead of sinking the code into the leading or trailing loop-nest (as is traditionally done in code-sinking), we sink all loop-nests that have the same set of outer loops into each other.

3.3 Redundant Dimensions in 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 7, statements

in the body of the transformed code are executed only when `t1 = t2`, so it is possible to eliminate the `t2` loop entirely, replacing all occurrences of `t2` in the body by `t1`. Therefore, dimension t_2 of the product space is redundant.

In general, we can determine redundant dimensions as follows. Affine embedding functions can be decomposed into their linear and offset parts as follows: $F_k(\vec{\imath}_k) = G_k\vec{\imath}_k + g_k$. We allow symbolic constants in the offset part of the embedding functions.

Theorem 9 *Let $\mathcal{P}$ be any Cartesian space and let $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ be a set of affine embedding functions $F_k : \mathcal{S}_k \rightarrow \mathcal{P}$. Let $F_k(\vec{\imath}_k) = G_k\vec{\imath}_k + g_k$. The number of independent dimensions of the space $\mathcal{P}$ is equal to the rank of matrix $G = [G_1 G_2 \ldots G_n]$.*

Proof: Trivial, hence omitted. □

For the embeddings of Figure 7, the matrix G is

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

where $G = [G_1 G_2]$. In this case, the rank of the matrix is 5 since the third row (corresponding to t_2) is the same as the first row.

Let p be the sum of the number of dimensions in all statement iteration spaces. Theorem 9 tells us that affine embedding functions cannot utilize more than p dimensions. We therefore use a p-dimensional space to model program execution orders.

There are transformations that use all dimensions of the product space. For example, embedding functions for modeling completely fissioned codes like the one in Figure 8 need all dimensions of the product space. In this case, the matrix G is full rank (in fact, it is the identity matrix $I_{6\times 6}$). Since we do not want to restrict transformations unnecessarily, we work with the full product space. Once

all embedding functions are determined, redundant dimensions are easy to identify and our algorithm removes them, so there is no performance penalty in the generated code.

While the independent dimensions of the product space are the ones that model program transformations, the affine part of dependent dimensions can reorder the execution order of statement instances mapped to the same point in outer dimensions. We remove dependent dimensions only when that is safe, as defined below.

Definition 10 *A dimension k of the product space is* redundant *if it satisfies the following properties:*

1. *Row k of the matrix G is a linear combination of rows $1, \ldots, (k-1)$; and*

2. *Removing dimension k of the product space does not violate any dependences. (I.e. removing the kth dimension of all difference vectors keeps them lexicographically non-negative.)*

3.4 Enhancing Locality in the Product Space

Consider a reuse class $\mathcal{R}$ and a reuse pair $(\vec{\imath}_s, \vec{\imath}_d) \in \mathcal{R}$. The abstract locality enhancement model in Section 2.4.2 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 be-

tween $F_s(\vec{\imath}_s)$ 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. However, since the product space is in effect a perfectly-nested loop, we can adapt the approach used for such loops to our context.

Consider the *reuse vector* ($\vec{v}$) for the reuse pair $(\vec{\imath}_s, \vec{\imath}_d)$ for a given choice of embedding functions $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$; we will refer to the jth 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.

4 An Example

[Figure 10 about here.]

Before presenting the general locality enhancement algorithm, we illustrate our approach on the imperfectly-nested matrix multiplication program in Figure 10. 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.

As shown in Figure 3, 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 transformed product space can be viewed as a perfectly-nested loop nest which has some number of *bands* of fully permutable loops; loops within the same band can be permuted in any order, while loops in different bands may not be permutable with each other.

The most difficult steps are (1) and (2), and these are interleaved in the algorithm described in Section 5. 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 5 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, (ii) permit sets of loops carrying reuse to be tiled, and (iii) 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 5 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*.

4.1 First Dimension

We first find embedding coefficients for the first dimension i_1 of the product space.

Legality

At the very least, the embeddings must not violate legality. From Definition 3, it follows that 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, as described in the appendix.

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

Minimizing Reuse Distance

System (1) 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 (1) 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.

4.2 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 tiling loops carrying reuse.

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

Legality

Generalizing the corresponding notion in perfectly-nested loops, we say that a dependence class $\mathcal{D} : D \begin{bmatrix} \vec{\imath}_s \\ \vec{\imath}_d \end{bmatrix} + d \geq 0$ 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 $\mathcal{D}$,

$$F_d^j(\vec{\imath}_d) - F_s^j(\vec{\imath}_s) = \begin{bmatrix} -G_s^j & G_d^j \end{bmatrix} \begin{bmatrix} \vec{\imath}_s \\ \vec{\imath}_d \end{bmatrix} + g_d^j - g_s^j \geq 0 \qquad (2)$$

In our running example, it can be shown that neither of the dependence classes $\mathcal{D}_1$ and $\mathcal{D}_2$ is satisfied by the first dimension of the embedding functions deter-

mined above, so both dependence classes must be considered when processing the second dimension.

Tiling Considerations

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 (2) holds even for satisfied dependence classes.

If the resulting system has no solutions, the current dimension cannot be made permutable with outer dimensions, so constraint (2) is dropped for satisfied dependence classes, and a new fully permutable band of loops is started at dimension j.

Minimizing Reuse Distance

From the solutions to the linear system that arises from legality and tiling considerations, 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).

4.3 Transformed Code

Our algorithm produces the following embeddings for the running example:

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

These embeddings allow all five dimensions to be tiled, so there is a single band of fully permutable loops.

Code Generation

Given these embeddings, the code generation algorithm uses Definition 10 to identify and eliminate redundant dimensions. The matrix G for the given embeddings is

$$\begin{array}{r} i_1 : \\ j_1 : \\ i_2 : \\ j_2 : \\ k_2 : \end{array} \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

It is easy to see that dimensions i_2 and j_2 are redundant and can be eliminated.

The remaining dimensions are tiled and appropriate code is generated. The resulting tiled code is shown in Figure 11. The *min*'s, *max*'s and conditionals within the loop body are removed by the code generation process using standard techniques from polyhedral algebra.

[Figure 11 about here.]

4.4 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) reuse distance minimization.

Figure 12 shows the high level structure of the algorithm. The algorithm processes the dimensions of the product space in order; when dimension j is processed, it does the following.

1. The algorithm first tries to find embedding co-efficients for dimension j, constrained only by legality. This is accomplished by the call to routine *LegalConstraints*, shown in Figure 13, which generates a linear system $Legal$ that constrains the jth dimension entry of difference vector $F_d(\vec{\imath}_d) - F_s(\vec{\imath}_s)$ to be non-negative for every $(\vec{\imath}_s, \vec{\imath}_d)$ in an *unsatisfied* dependence class.

2. If this system has solutions, the algorithm tries to find embedding co-efficients that would permit dimension j to be added to the current band of fully permutable loops. This is accomplished by the call to the routine $SimpleTilingConstraints$, shown in Figure 14, which additionally constrains the jth dimension entry of difference vector $F_d(\vec{\imath}_d) - F_s(\vec{\imath}_s)$ to be non-negative for every $(\vec{\imath}_s, \vec{\imath}_d)$ in a *satisfied* dependence class.

   If it succeeds, dimension j is added to the current fully permutable band; otherwise, a new band is started.

3. From the set of possible embeddings determined by the previous step, the algorithm picks an embedding that enhances locality. This is accomplished by the call to routine $PromoteReuse$ which uses a sequence of reuse classes ranked by importance to determine embedding co-efficients that enhance locality. In our implementation, we rank reuse classes by estimating the number of reuse pairs in each class.

The code generation step eliminates redundant dimensions, tiles loops that carry reuse, and generates code.

By introducing tiling constraints in the second step before promoting reuse for particular reuse classes in the third step, we have in effect given precedence

to tiling over promoting reuse for reuse classes. In other words, the algorithm for promoting reuse is constrained to pick co-embedding efficients from the set of co-efficients that are known to be both legal, and desirable for tiling. It is possible to constrain co-efficients in the other order to give precedence to promoting reuse for particular reuse classes, but we believe our design choice leads to better performance.

[Figure 12 about here.]

[Figure 13 about here.]

[Figure 14 about here.]

[Figure 15 about here.]

5 Algorithm for Locality Enhancement

We now present the complete algorithm for locality enhancement which simultaneously determines the embeddings F_i and the transformation `T` of the product space.

5.1 Product Space Transformations

Since the product space can be viewed as a perfectly-nested loop, the key transformations for this space are unimodular transformations—namely, permutation, skewing, and reversal.

To see the need for these transformations, consider Algorithm `SimpleLocalityEnhancement` which was introduced in Section 4. This algorithm processes the dimensions of the product space in order. When processing dimension j, we may not be able to find embedding co-efficients that permit us to add loop j to the current band of fully permutable loops; if so, we make loop j the first loop in a new band of fully permutable loops. However, it is possible that *reversing* loop j (that is, scanning the points in the product space in reverse lexicographic order along dimension j) is legal, and this may permit us to add this loop to the current band of fully permutable loops. Therefore, if the linear system S in Algo-

rithm `SimpleLocalityEnhancement` does not have a solution, we should try loop reversal of dimension j before giving up and terminating the current fully permutable band of loops.

This strategy can be implemented as follows. We first construct a system S which constrains the jth dimension entry of difference vector $F_d(\vec{\imath}_d) - F_s(\vec{\imath}_s)$ to be non-negative for every $(\vec{\imath}_s, \vec{\imath}_d)$ in both unsatisfied and satisfied dependence classes. If this system does not have a solution, we construct a new system S' which constrains the jth dimension entry of difference vector $F_d(\vec{\imath}_d) - F_s(\vec{\imath}_s)$ to be *non-positive* for every $(\vec{\imath}_s, \vec{\imath}_d)$ in both unsatisfied and satisfied dependence classes. If S' has solutions, we can use them to find appropriate embedding coefficients for dimension j, provided we remember to reverse the direction of loop j when we generate code.

Skewing of the product space can be incorporated into the algorithm in a similar manner. Dimension j may be permutable *after skewing* by outer dimensions if the difference vector entries corresponding to the satisfied dependence classes are bounded below by a negative constant. Hence, for every pair $(\vec{\imath}_s, \vec{\imath}_d)$ in all satisfied

dependence classes $\mathcal{D}$, we require that

$$\begin{bmatrix} -G_s^j & G_d^j \end{bmatrix} \begin{bmatrix} \vec{\imath}_s \\ \vec{\imath}_d \end{bmatrix} + g_d^j - g_s^j + \alpha \geq 0, \quad \alpha \geq 0,$$

where α is an additional variable introduced into the system. The solution with the smallest α allows us to make dimension j permutable with the outer dimensions using a minimum amount of skewing. If α can be chosen to be 0, the dimension is permutable with outer dimensions without skewing.

Finally, permutation of product space dimensions can be incorporated into the algorithm as follows. Suppose dimension j is not fully permutable with the current band of fully permutable loops even after skewing and/or reversal. Instead of giving up, we can try to permute j with a dimension k ($j < k \leq |\mathcal{P}|$) of the product space for which we can find suitable embedding functions.

Figure 16 shows the complete locality enhancement algorithm in which the determination of embedding coefficients is interleaved with the determination of the product space transformation as described above. Each iteration of the outermost `while` loop tries to construct one dimension of the transformed product space. The inner `for-each-`q loop examines unmapped dimensions of the product space to find one which can be made the next dimension of the transformed

product space and appended to the current band of fully permutable loops after skewing and/or reversal if necessary. If such a dimension is found, procedure `PromoteReuse` in Figure 15 is called 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. If no legal embeddings can be found for any unmapped dimension even after this, the algorithm fails. The algorithm terminates successfully when all dimensions of the product space have been mapped into the transformed space.

In our experiments, the algorithm has always terminated successfully. We conjecture that it can always find embeddings consistent with Definition 5, but we have not proved this.

[Figure 16 about here.]

[Figure 17 about here.]

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 techniques in the literature can be used. Here we present a simple heuristic similar to *memory order*.[16] 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 $\{F_1^j, F_2^j, \dots, F_n^j\}$ to be the number of reuse pairs in the classes for 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 17 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 10, our algorithm places all five dimensions of the product space in a single fully permutable band. It then picks the dimension order $j_1 \times j_2 \times k_2 \times i_1 \times i_2$.

5.2 Code Generation

Removing Redundant Dimensions

Let $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ be the set of embedding functions and let $F_k(\vec{\imath}_k) = G_k\vec{\imath}_k + g_k$. As discussed in Section 3, any dimension j for which the jth row G^j of the matrix $G = [G_1, G_2, \ldots, G_n]$ is linearly dependent on other rows is redundant as long as its removal does not affect dependences.

Once a good legal ordering of the dimensions is determined, our algorithm identifies all dimensions j for which G^j is linearly dependent on previous dimensions, and eliminates those dimensions that do not affect dependences.

Tiling

All dimensions with non-zero $ReusePenalty$ will benefit from tiling. Each fully permutable band is tiled (after individual dimensions are skewed by outer loops if

necessary).

For the running example, j_1, k_2, and i_1 are not redundant and exhibit reuse, therefore we decide to tile all of them. They are all in the same band and do not require skewing.

5.3 Tile Size Determination

We use simple heuristics to determine tile sizes.

We estimate the *data foot-print* (the total amount of data touched by a tile), and require that it fit into the memory hierarchy level under 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 for the tile size we compute).

Our algorithm 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. Since data references are affine, this enables us to calculate the data accessed by each point in the product space.

   In the running example of Figure 10, a point $(x_1, x_2, x_3, x_4, x_5)$ of our

transformed product space has 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 (that is, if the linear parts of the two references are the same).

   There are three equivalence classes $\{S_2 : a(x_4, x_3)\}$, $\{S_2 : b(x_1, x_3)\}$ and $\{S_1 : c(x_4, x_1), S_2 : c(x_4, x_1)\}$, in our example.

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 the literature.(27) More accurate solutions can be obtained by using Erhart Polynomials.(8)

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,[9,11,20] but we do not use them in our current implementation.

6 Experimental Results

In this section, we present results from our implementation for four important codes. 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). Wherever possible, we present three sets of performance numbers for a 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.[32]

2. Performance of code produced by an implementation of the techniques described in this paper.

   These codes were compiled by the SGI MIPSPro compiler with the flags "-O3 -LNO:blocking=off" to disable blocking by the SGI compiler.

3. Performance of hand-coded LAPACK library routine running on top of hand-tuned BLAS.

Performance is reported in MFLOPS, counting each multiply-add as 1 Flop. For some of the codes like tomcatv, we did not have hand-coded versions as a comparison; in these cases, we report running time.

6.1 Cholesky Factorization

Cholesky factorization is used to solve symmetric positive-definite linear systems. Figure 18 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 19(a) compares the performance of all six versions compiled by the SGI compiler, the hand-optimized LAPACK library rou-

tine, 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.

[Figure 18 about here.]

[Figure 19 about here.]

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 all six versions of Cholesky factorization.

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

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

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 19(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.

Figure 19(c) compares the performance of this code (tiled for two levels of the memory hierarchy) with the performance of our code tiled for a *single* level of the memory hierarchy but using a range of tile sizes from 10 to 250. The size of the array on which Cholesky factorization is done is 2000×2000. Our experiments show that two-level blocking gives better performance.

6.1.1 Triangular Solve

Triangular systems of equations of the form `Lx = b` where `L` is a lower triangular matrix, `b` is a known vector and `x` is the vector of unknowns arise frequently

in applications. Sometimes, it is necessary to solve multiple triangular systems that have the same co-efficient matrix `L`. Such multiple systems can obviously be viewed as computing a matrix `X` that satisfies the equation `LX = B` where `B` is a matrix whose columns are constituted from the right-hand sides of all the triangular systems. The code in Figure 20 solves such multiple triangular systems, overwriting `B` with the solution.

[Figure 20 about here.]

[Figure 21 about here.]

For this code, our algorithm finds the following embeddings:

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

The fourth and fifth dimensions of the product space were redundant, so they were eliminated. Our algorithm decides to tile all three remaining dimensions. It chooses a tile size of 36 for the L1 cache and 288 for the L2 cache for all the dimensions.

Figure 21(a) shows performance results for a constant number of right-hand sides (`M` in Figure 20 is 100). The performance of code generated by our techniques is up to 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 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 21(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.

6.2 Jacobi

[Figure 22 about here.]

Our next benchmark is the Jacobi kernel in Figure 1, which was discussed in Section 1. 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 improves reuse and spatial locality of the arrays in the two 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 algorithm chooses a tile size of 20 for the L1 cache and 160 for the L2 cache. The generated code is shown in Figure 25 in the appendix.

Figure 22(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). Comparison with other tile sizes is shown in Figure 22(b).

6.3 Tomcatv

[Figure 23 about here.]

[Figure 24 about here.]

As a final example, we consider the tomcatv code from the SPECfp benchmark suite. The code (Figure 23) 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 24(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[5], 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 24(b).

6.4 Discussion

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

[5]The resulting kernel can be tiled speculatively as demonstrated by Song and Li.(30)

to optimize our benchmarks, it does not find the right sequence of transformations, so the performance of the resulting code suffers. Furthermore, for Cholesky factorization, the performance of our optimized code approaches the performance of hand-written library code in LAPACK. For triangular solve, we generate code with the same high-level structure as library code; we believe that tuning register allocation and instruction scheduling in the MIPSPro compiler will allow this code to perform as well as library code. Finally, the advantage of our general-purpose technique is that it can be used for codes like Jacobi and tomcatv for which the LAPACK library is not useful.

7 Related Work and Conclusions

The work reported in this paper grew out of our effort to design methods for automatic locality enhancement of codes that are important in computational science. Much of computational science is concerned with the numerical solution of partial differential equations. Solution techniques are classified into *implicit* and *explicit* methods. Implicit methods require the solution of large systems of linear algebraic equations; Cholesky factorization is usually employed for this purpose. The Jacobi code on the other hand is an example of an explicit method; in these meth-

ods, the value of the dependent variable of the pde at some step of the computation is computed as a simple function of the value of that variable in previous steps. As we have shown, the locality enhancement techniques described in this paper can be used for both classes of methods.

The mathematical techniques used in this paper have been used by the systolic array community for scheduling statements in loop nests on systolic arrays.(19) These techniques were extended by Feautrier in his theory of *schedules* in multi-dimensional time(10) which he used for automatic parallelization; related approaches are mappings and affine transforms.(14, 23)

Our approach generalizes techniques used in current compilers for locality enhancement of both perfectly-nested and imperfectly-nested loops.(5, 22, 31–33) The use of embeddings and the product space generalizes techniques like statement sinking and loop fusion that are used in compilers such as the SGI MIPSPro compiler to convert some imperfectly-nested loops into perfectly-nested loops. The particular approach we have taken to locality enhancement is a generalization of the approach of Li and Pingali(22) which was developed for perfectly-nested loops.

For enhancing locality in imperfectly-nested loops, Kelly and Pugh advocate searching the space of legal transformations using cost models that evaluate the mappings produced.(15) They associate a multi-dimensional mapping with each

statement but the range of these mappings is not fixed since there is no analog of our product space. Starting from totally unspecified mappings, they propose to explore the tree of partially specified mappings till they completely specify the mapping, using estimates of the number of cache misses to guide the search. Their estimator targets only reuses with source and destination in the same statement. In particular, reuse between different statements requiring fusion will not be modeled. Though they represent tiling by means of pseudo-linear functions (using `mod` and `div`), they do not include them in their search space of possible transformations and hence do not necessarily find solutions that can be tiled. By using a well-defined space (the product space), we are able to target all reuses by representing reuse distances between statement instances by reuse vectors which we can then minimize dimension by dimension. Furthermore, the product space allows us to impose constraints so that dimensions are permutable—this lets us order the dimensions and tile them to reduce reuse distances further.

A different approach to locality enhancement of imperfectly-nested loops has been taken in *data-centric approaches* such as data shackling.[(17)] The compiler determines an order in which array elements should be touched, and then schedules code so that all statements that touch a given data element are scheduled to execute when that data item is brought into the cache. Integer linear program-

ming techniques are used to determine if such a schedule is legal. This work has been extended by Pugh and Rosser in their work on iteration space slicing which permits them to synthesize legal data-centric schedules.[28] The data-centric approach can be used to generate code for sparse matrix applications as well.[26] The framework in this paper can be used to generate data-centric code by adding data dimensions to the product space.[1]

There is considerable interest in the numerical analysis community in developing *block-recursive* codes for good performance on multi-level memory hierarchies.[7,13] Block-recursive codes can be viewed as divide-and-conquer style codes in which the original problem is repeatedly sub-divided until the working set fits into the highest level of the memory hierarchy. In a certain sense, these codes can be viewed as being automatically blocked for all levels of the memory hierarchy. We have recently shown that our framework can be used to derive block-recursive codes automatically from iterative codes.[2] A data-centric approach to this problem has been explored by Yi et al.[34]

In future work, we would like to explore the scalability of our techniques for programs with large numbers of imperfectly-nested loop nests. Another line of research is motivated by the observation that dependence analysis appears to be inadequate to permit automatic restructuring of some codes such as LU factor-

ization with pivoting. Respecting dependences is a sufficient but not necessary condition for the legality of a transformation; for example, the reassociation of a reduction operation violates dependences but is legal nevertheless because the original program and the restructured program produce the same answers. Similarly, it can be shown that restructuring LU with pivoting to enhance locality violates dependences but is legal nevertheless because it relies on the fact that row permutations commute with updates.(12) We have recently developed a novel form of symbolic analysis called *fractal symbolic analysis*(24,25) to address this problem. Although fractal symbolic analysis can be used to *verify* the correctness of these transformations, we do not yet have a framework like the one in this paper that would enable a compiler to *synthesize* good sequences of transformations. The development of such a framework is an important open problem in compiler research.

References

[1] Nawaaz Ahmed, Nikolay Mateev, Keshav Pingali, and Paul Stodghil. A framework for sparse matrix code synthesis from high-level specifications. In *Proceedings of SC2000*, Dallas, Texas, November 2000.

[2] Nawaaz Ahmed and Keshav Pingali. Automatic generation of block-recursive codes. In *Proceedings of Euro-Par*, Munich, Germany, August 29 – September 1, 2000.

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

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

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

[6] Steve Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Supercomputing*, 1992.

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

[9] Stephanie Coleman and Kathryn 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.

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

[11] 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.

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

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

[14] Wayne Kelly and William Pugh. Finding legal reordering transformations using mappings. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 107–124, Ithaca, New York, August 8–10, 1994. Springer-Verlag.

[15] Wayne Kelly and William Pugh. Selecting affine mappings based on performance estimation. *Parallel Processing Letters*, 4(3):205–209, September 1994.

[16] 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.

[17] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *Programming Languages, Design and Implementation*. ACM SIGPLAN, June 1997.

[18] Induprakas Kodukula, Keshav Pingali, Robert Cox, and Dror Maydan. Imperfectly nested loop transformations for memory hierarchy management. In *International Conference on Supercomputing*, Rhodes, Greece, June 1999.

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

[20] Monica S. Lam, Edward E. Rothberg, and Michael 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, April 8–11, 1991.

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

[22] Wei Li and Keshav Pingali. A singular loop transformation based on nonsingular matrices. *International Journal of Parallel Programming*, 22(2), April 1994.

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

[24] Nikolay Mateev, Vijay Menon, and Keshav Pingali. Left-looking to right-looking and vice versa: An application of fractal symbolic analysis to linear

algebra code restructuring. In *Proceedings of Euro-Par*, Munich, Germany, August 29 – September 1, 2000.

[25] Nikolay Mateev, Vijay Menon, and Keshav Pingali. Fractal symbolic analysis for program transformations. In *ACM International Conference on Supercomputing (ICS) 2001*. ACM, June 2001.

[26] Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *Proceedings of the 2000 International Conference on Supercomputing*, Santa Fe, New Mexico, May 8–11, 2000.

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

[28] William Pugh and Evan Rosser. Iteration space slicing for locality. In *Proc. of 12th International Workshop on Languages and Compilers for Parallel Computing, (LCPC99)*, August 1999.

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

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

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

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

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

[34] Qing Yi, Vikram Adve, and Ken Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *Proceedings of the 2000 ACM Symposium on Programming Language Design and Implementation*, Vancouver, Canada, June 18–21, 2000.

A Farkas' Lemma

We show how to apply Farkas' Lemma to determine constraints on embeddings co-efficients.

Consider a dependence class $\mathcal{D} : D \begin{bmatrix} \vec{\imath}_s \\ \vec{\imath}_d \end{bmatrix} + d \geq 0$, and a dependence pair $(\vec{\imath}_s, \vec{\imath}_d) \in \mathcal{D}$. Let F_d and F_s be the embedding functions for the destination and source statements of this dependence, and let the jth dimensions of these functions be $F_d^j = G_d^j \vec{\imath}_d + g_d^j$ and $F_s^j = G_s^j \vec{\imath}_s + g_s^j$.

Suppose that we must choose these unknown coefficients (G, g) so that $F_d^j(\vec{\imath}_d) - F_s^j(\vec{\imath}_s) \geq 0$.

The affine form of Farkas' lemma lets us express the constraints on these co-efficients in terms of dependence class coefficients (D, d).

Lemma 1 *(Farkas' Lemma) 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:*

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

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

$$\begin{bmatrix} -G_s^j & G_d^j \end{bmatrix} \begin{bmatrix} \vec{i}_s \\ \\ \vec{i}_d \end{bmatrix} + g_d^j - g_s^j = \lambda_0 + \Lambda^T D \begin{bmatrix} \vec{i}_s \\ \\ \vec{i}_d \end{bmatrix} + \Lambda^T d, \qquad \lambda_0 \geq 0, \Lambda \geq 0.$$

Equating coefficients of $\vec{i}_s$ and $\vec{i}_d$ on both sides, we get

$$\begin{aligned} \begin{bmatrix} -G_s^j & G_d^j \end{bmatrix} &= \Lambda^T D, \\ g_d^j - g_s^j &= \lambda_0 + \Lambda^T d, \\ \lambda_0 \geq 0, \quad & \quad \Lambda \geq 0. \end{aligned}$$

The Farkas multipliers can be eliminated through Fourier-Motzkin projection to give a system of inequalities constraining the unknown embedding coefficients.

As an example, consider the first dimension of the embedding functions for the running example in Section 4. The following conditions must be satisfied:

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 $(i_1, j_1, i_2, j_2, k_2)$ in $\mathcal{D}_1 = \{(i_1, j_1, i_2, j_2, k_2) : 1 \leq i_1, j_1, i_2, j_2, k_2 \leq N, i_1 = i_2, j_1 = j_2\}$, 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 $(i_2, j_2, k_2, i'_2, j'_2, k'_2)$ in $\mathcal{D}_2 = \{(i_2, j_2, k_2, i'_2, j'_2, k'_2) : 1 \leq i_2, j_2, k_2, i'_2, j'_2, k'_2 \leq N, i_2 = i'_2, j_2 = j'_2, k_2 < k'_2\}$.

Let us apply Farkas' lemma to the first condition. We have

$$
\begin{aligned}
&G^1_{i_2}i_2 + G^1_{j_2}j_2 + G^1_{k_2}k_2 + g^1_N N + g^1_1 - i_1 = \\
&\quad \lambda_0 + \lambda_1(i_1 - 1) + \lambda_2(N - i_1) + \lambda_3(j_1 - 1) \\
&\quad + \lambda_4(N - j_1) + \lambda_5(i_2 - 1) + \lambda_6(N - i_2) + \lambda_7(j_2 - 1) \\
&\quad + \lambda_8(N - j_2) + \lambda_9(k_2 - 1) + \lambda_{10}(N - k_2) \\
&\quad + \lambda_{11}(i_1 - i_2) + \lambda_{12}(i_2 - i_1) + \lambda_{13}(j_1 - j_2) \\
&\quad + \lambda_{14}(j_2 - j_1), \\
&\lambda_0, \lambda_1, \ldots, \lambda_{14} \geq 0
\end{aligned}
$$

After equating coefficients on both sides, we get:

$$\begin{aligned}
i_1: \quad -1 &= \lambda_1 - \lambda_2 + \lambda_{11} - \lambda_{12} \\
j_1: \quad 0 &= \lambda_3 - \lambda_4 + \lambda_{13} - \lambda_{14} \\
i_2: \quad G^1_{i_2} &= \lambda_5 - \lambda_6 + \lambda_{12} - \lambda_{11} \\
j_2: \quad G^1_{j_2} &= \lambda_7 - \lambda_8 + \lambda_{14} - \lambda_{13} \\
k_2: \quad G^1_{k_2} &= \lambda_9 - \lambda_{10} \\
N: \quad g^1_N &= \lambda_2 + \lambda_4 + \lambda_6 + \lambda_8 + \lambda_{10} \\
1: \quad g^1_1 &= \lambda_0 - \lambda_1 - \lambda_3 - \lambda_5 - \lambda_7 - \lambda_9
\end{aligned}$$

Eliminating Farkas multipliers through Fourier-Motzkin projection, we obtain

the following constraints on the unknown embedding coefficients:

$$\begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} G^1_{i_2} \\ G^1_{j_2} \\ G^1_{k_2} \\ g^1_N \\ g^1_1 \end{bmatrix} \geq \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad (3)$$

Going through the same steps for the second condition, we get

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} G^1_{i_2} \\ G^1_{j_2} \\ G^1_{k_2} \\ g^1_N \\ g^1_1 \end{bmatrix} \geq \begin{bmatrix} 0 \end{bmatrix} \quad (4)$$

Combining conditions (3) and (4), we obtain the system of inequalities shown

in Section 4:

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

B Generated code for Jacobi

[Figure 25 about here.]

The Jacobi code generated by our implementation tiled with a tile size of 20 is shown in Figure 25.

List of Figures

1 Jacobi 84
2 Fused Jacobi 85
3 Locality Enhancement of Imperfectly-nested Loop Nests 86
4 Modeling Program Execution 87
5 Embeddings and Code Generation 88
6 1-D Embeddings for Loop Jamming 89
7 A Perfectly-nested Version of Jacobi 90
8 Embeddings for Loop Fission 91
9 Embeddings for Loop Fusion 92
10 Imperfectly-nested MMM 93
11 Locality-optimized MMM before Code Simplification 94
12 Simple Algorithm to Enhance Locality 95
13 Formulating Linear System for Legality 96
14 Formulating Linear System for Tiling 97
15 Formulating Linear Systems for Promoting Reuse 98
16 Algorithm to Enhance Locality 99
17 Determining Dimension Ordering 100
18 kij-Cholesky Factorization: Original Code 101
19 Cholesky Factorization and its Performance 102
20 Triangular Solve: Original Code 103
21 Triangular Solve and its Performance 104
22 Jacobi and its Performance 105
23 Tomcatv Kernel 106
24 Performance of tomcatv 107
25 Generated code for Jacobi 108

```
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: Jacobi

```
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 2: Fused Jacobi

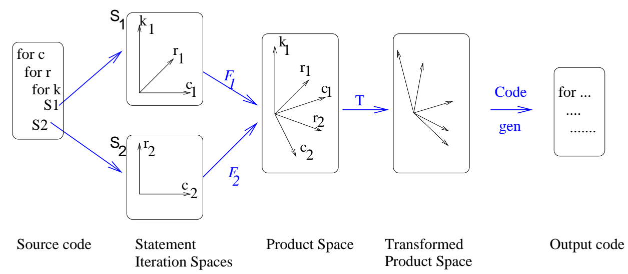
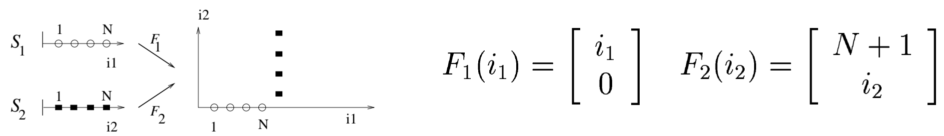


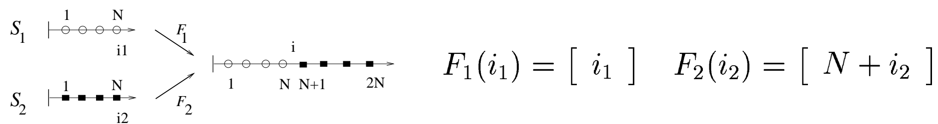
Figure 3: Locality Enhancement of Imperfectly-nested Loop Nests

```
for i1 = 1, N
S1: x[i1] = a[i1]
end
for i2 = 1, N
S2: x[i2] += b[i2]
end
```

(a) Code Fragment

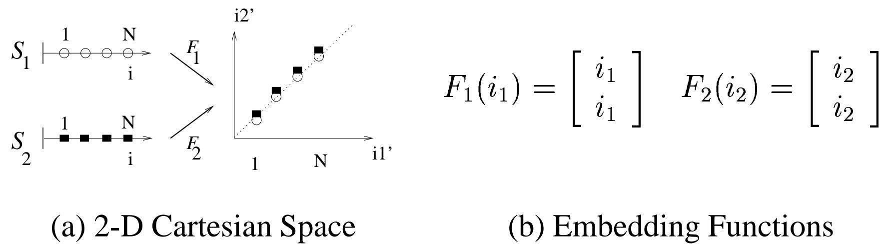


(b) 2-D Cartesian Space and Embeddings



(c) 1-D Cartesian Space and Embeddings

Figure 4: Modeling Program Execution



(a) 2-D Cartesian Space (b) Embedding Functions

```
//scan entire space in lexicographic order
for i1' = -∞, +∞
  for i2' = -∞, +∞
    //at each point, execute statement instances mapped there
    for i1 = 1, N
      S1:if ((i1 == i1')&&(i1==i2'))  x[i1] = a[i1]
    end
    for i2 = 1, N
      S2:if ((i2 == i1')&&(i2==i2'))  x[i2] += b[i2]
    end
  end
end
```

(c) Unoptimized Code

```
for i1' = 1, N
   x[i1'] = a[i1']
   x[i1'] += b[i1']
end
```

(d) Optimized Code

Figure 5: Embeddings and Code Generation

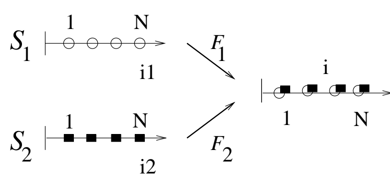


Figure 6: 1-D Embeddings for Loop Jamming

.

```
for t1 = 1, T
for i1 = 2, N-1
for j1 = 2, N-1
for t2 = 1, T
for i2 = 2, N-1
for j2 = 2, N-1
  if (t2 == t1 && i2 == 2 && j2 == 2)
S1:   L(i1,j1) = (A(i1,j1+1) + A(i1,j1-1)
                 + A(i1+1,j1) + A(i1-1,j1)) / 4
  endif
  if (t1 == t2 && i1 == N-1 && j1 == N-1)
S2:   A(i2,j2) = L(i2,j2)
  endif
end
end
end
end
end
end
```

Figure 7: A Perfectly-nested Version of Jacobi

```
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
end
for t = 1, T
  for i2 = 2, N-1
    for j2 = 2, N-1
S2:    A(i2,j2) = L(i2,j2)
    end
  end
end
```

(a) Fissioned code

$$F_1\left(\begin{bmatrix} t_1 \\ i_1 \\ j_1 \end{bmatrix}\right) = \begin{bmatrix} t_1 \\ i_1 \\ j_1 \\ 1 \\ 2 \\ 2 \end{bmatrix} \quad F_2\left(\begin{bmatrix} t_2 \\ i_2 \\ j_2 \end{bmatrix}\right) = \begin{bmatrix} T \\ N-1 \\ N-1 \\ t_2 \\ i_2 \\ j_2 \end{bmatrix}$$

(b) Embeddings

Figure 8: Embeddings for Loop Fission

```
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
S2:   A(i,j) = L(i,j)
    end
  end
end
```

(a) Fused code

$$F_1(\begin{bmatrix} t_1 \\ i_1 \\ j_1 \end{bmatrix}) = \begin{bmatrix} t_1 \\ i_1 \\ j_1 \\ t_1 \\ i_1 \\ j_1 \end{bmatrix} \quad F_2(\begin{bmatrix} t_2 \\ i_2 \\ j_2 \end{bmatrix}) = \begin{bmatrix} t_2 \\ i_2 \\ j_2 \\ t_2 \\ i_2 \\ j_2 \end{bmatrix}$$

(b) Embeddings

Figure 9: Embeddings for Loop Fusion

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

Figure 10: Imperfectly-nested MMM

```
//tile counter loops
for t1 = 1, N, B
for t2 = 0, 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)
        if (k == 0)
          c(i,j) = 0
        endif
        c(i,j) = c(i,j) + a(i,k) * b(k,j)
```

Figure 11: Locality-optimized MMM before Code Simplification

```
ALGORITHM SimpleLocalityEnhancement

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);

for j := 1 to | P| //for each dimension of product space
      L = LegalityConstraints(j, DU);
      if system L has solutions
         S = SimpleTilingConstraints(j,L,DS);
         if system S has solutions
             //add dimension to current fully permutable band
             Embedding coefficients for dimension j =
                      PromoteReuse(j,S,RS);
         else
             // No more dimensions can be added to current band.
             // Start a new band of fully permutable loops.
             DS := empty set;
             Embedding coefficients for dimension j =
                  PromoteReuse(j,L,RS);
         Update DS and DU;

     else fail;
end
Eliminate redundant dimensions;
Tile permutable dimensions with non-zero ReusePenalty;
```

Figure 12: Simple Algorithm to Enhance Locality

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

  Legal = System constraining F_d(i_d)[q] - F_s(i_s)[q] to be non-negative
              for every (i_s, i_d) in dependence class in DU;

  Use Farkas' lemma to convert system Legal into
    a system L constraining unknown embedding
    coefficients;

  Return L;

end
```

Figure 13: Formulating Linear System for Legality

```
ALGORITHM SimpleTilingConstraints(q, L,  DS )
 /*
   q is dimension being processed.
   L is a system constraining embedding coefficients.
   DS is set of satisfied dependence classes.
 */

  Sat = System constraining F_d(i_d)[q] - F_s(i_s)[q]
            to be non-negative for every (i_s, i_d) in dependence class in DS;

  Use Farkas' lemma to convert conjunction of L and Sat
  to a system S constraining unknown embedding coefficients;

  Return S;
end
```

Figure 14: Formulating Linear System for Tiling

```
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.
 */

  L':= L
  for every reuse class R in RS in priority order
      Z := System constraining q^th dimension of
           reuse vectors in class R to be zero;

      if (conjunction of L' and Z has solution)
         L' := conjunction of L' and Z;
      endif
  endfor

  Return any set of coefficients satisfying L';
end
```

Figure 15: Formulating Linear Systems for Promoting Reuse

```
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);
T  := Transformation matrix for product space
           (initialized to identity);

while (Q is non-empty)
   for each q in Q
       Construct a system S constraining
           . F_d(i_d)[q] - F_s(i_s)[q] to be non-negative
             for every (i_s,i_d) in dependence class in DU;
           . F_d(i_d)[q] - F_s(i_s)[q] + positive α to be non-negative
             for every (i_s,i_d) in dependence class in DS;
       if system S has solutions
            Embedding coefficients for dimension j =
               PromoteReuse(q,L,RS);
            Update DS, DU and T;
            Delete q from Q;
            j = j + 1;
            continue while-loop;
       endif;

       //try loop reversal
       Construct a system S' constraining
           . F_d(i_d)[q] - F_s(i_s)[q] to be non-positive
             for every (i_s,i_d) in dependence class in DU;
           . F_d(i_d)[q] - F_s(i_s)[q] - positive α to be non-positive
             for every (i_s,i_d) in dependence class in DS;
       if system S' has solutions
            Embedding coefficients for dimension j =
               PromoteReuse(q,L,RS);
            Update DS, DU and T;
            Delete q from Q;
            j = j + 1;
            continue while-loop;
       endif;
   endfor;

   //if we reach here, we cannot find a dimension to add to current band
   //of fully permutable loops.
   if (DS == {}) //no legal embeddings
      fail;
   else  //start a new band of fully permutable loops
      DS = {};
endwhile;

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

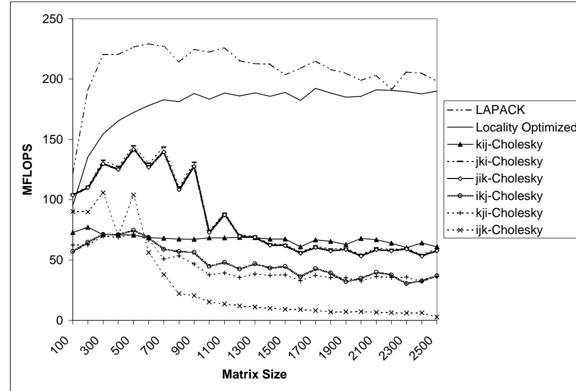
Figure 16: Algorithm to Enhance Locality

```
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 = {i1', i2', ... ik'}
        if {i1', i2', ... ik', l} is legal
           NRPO = {i1', i2', ... ik', l}
            RPO  = RPO − {l}
            m = m − 1
            k = k + 1
            continue while loop
        endif
    endfor
endwhile
```

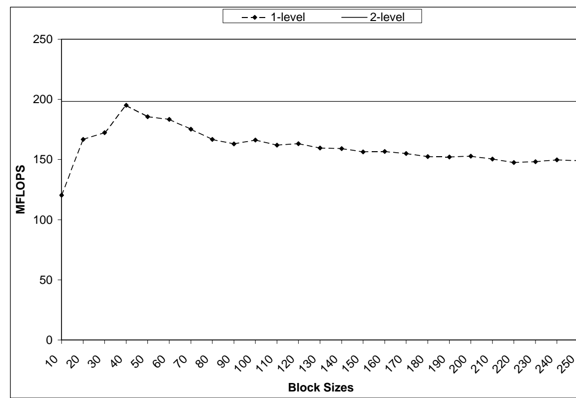
Figure 17: Determining Dimension Ordering

```
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 18: kij-Cholesky Factorization: Original Code



(a) Performance

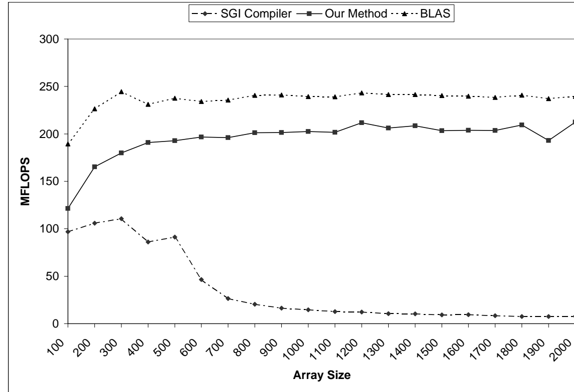


(b) Variation with tile size

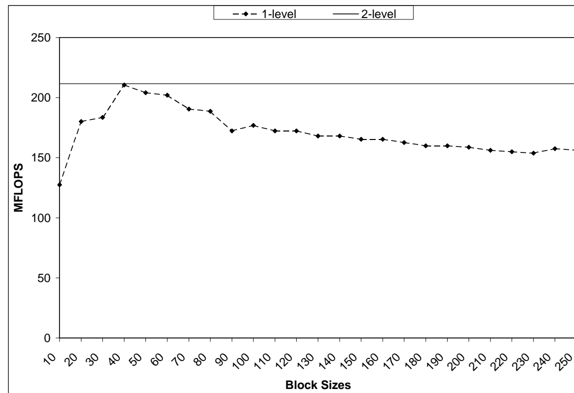
Figure 19: Cholesky Factorization and its Performance

```
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 20: Triangular Solve: Original Code

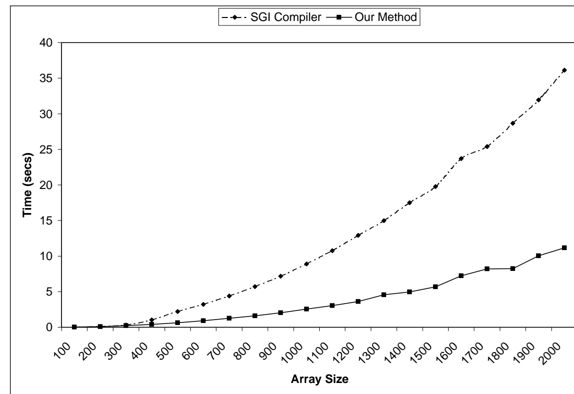


(a) Performance

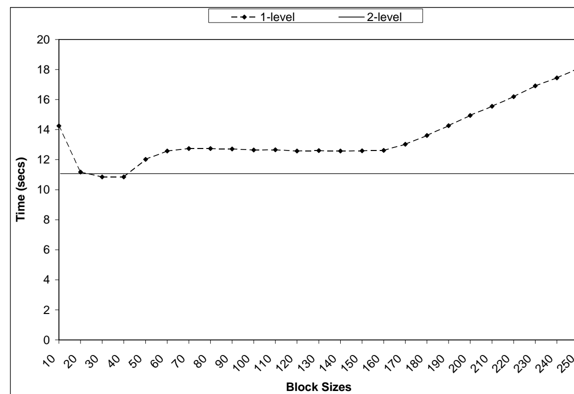


(b) Variation with tile size

Figure 21: Triangular Solve and its Performance



(a) Performance

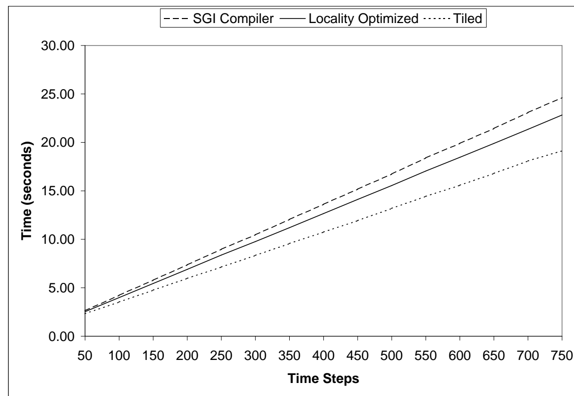


(b) Variation with tile size

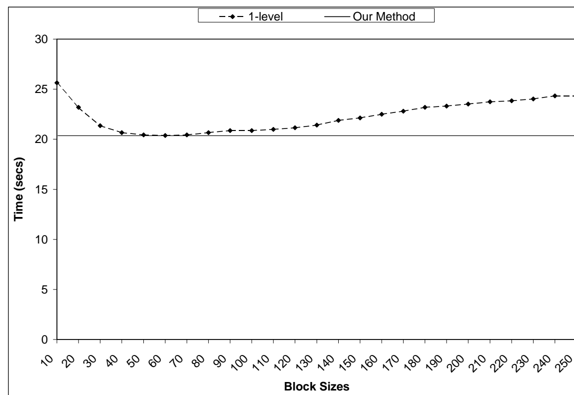
Figure 22: Jacobi and its Performance

```
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     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 23: Tomcatv Kernel



(a) Performance



(b) Variation with tile size

Figure 24: Performance of tomcatv

```
if (N .GE. 3) then
   do b_t = 1, (T+19)/20
      do b_j = 2*b_t-1, min((40*b_t+N+19)/20,(N+2*T+19)/20)
         b_il = max((-N+20*b_j+3)/20,2*b_t-1)
         b_iu = min((40*b_t+N+19)/20,(N+20*b_j+16)/20,(N+2*T+19)/20)
         do b_i = b_il, b_iu
            tl = max((-N+20*b_j-18)/2,(-N+20*b_i-18)/2,20*b_t-19)
            tu = min(10*b_i-1,20*b_t,T,10*b_j-1)
            do t = tl, tu
               do j = max(20*b_j-2*t-19,2), 2
                  do i = max(20*b_i-2*t-19,2), min(N-1,20*b_i-2*t)
                     L(i,j) = (A(i,j+1)+A(i,j-1)+A(i+1,j)+A(i-1,j))/4
                  enddo
               enddo
               do j = max(20*b_j-2*t-19,3), min(N-1,20*b_j-2*t)
                  do i = max(20*b_i-2*t-19,2), 2
                     L(i,j) = (A(i,j+1)+A(i,j-1)+A(i+1,j)+A(i-1,j))/4
                  enddo
                  do i = max(20*b_i-2*t-19,3), min(N-1,20*b_i-2*t)
                     L(i,j) = (A(i,j+1)+A(i,j-1)+A(i+1,j)+A(i-1,j))/4
                     A(i-1,j-1) = L(i-1,j-1)
                  enddo
                  do i = N, min(N,20*b_i-2*t)
                     A(i-1,j-1) = L(i-1,j-1)
                  enddo
               enddo
               do j = N, min(N,20*b_j-2*t)
                  do i = max(20*b_i-2*t-19,3), min(N,20*b_i-2*t)
                     A(i-1,j-1) = L(i-1,j-1)
                  enddo
               enddo
            enddo
         enddo
      enddo
   enddo
endif
```

Figure 25: Generated code for Jacobi