Transforming Complex Loop Nests For Locality

Qing Yi[†] Ken Kennedy[†] Vikram Adve[‡]

[†] Rice University, 6100 Main Street MS-132, Houston, TX 77005

[‡] University of Illinois at Urbana-Champaign, 1304 W. Springfield Ave, Urbana, IL 61801

Abstract

Over the past 20 years, increases in processor speed have dramatically outstripped performance increases for standard memory chips. To bridge this gap, compilers must optimize applications so that data fetched into caches are reused before being displaced. Existing compiler techniques can efficiently optimize simple loop structures such as sequences of perfectly nested loops. However, on more complicated structures, existing techniques are either ineffective or require too much computation time to be practical for a commercial compiler.

To optimize complex loop structures both effectively and inexpensively, we present a novel loop transformation, *dependence hoisting*, for optimizing arbitrarily nested loops, and an efficient framework that applies the new technique to aggressively optimize benchmarks for better locality. Our technique is as inexpensive as the traditional *unimodular* loop transformation techniques and thus can be incorporated into commercial compilers. In addition, it is highly effective and is able to block several linear algebra kernels containing highly challenging loop structures, in particular, Cholesky, QR, LU factorization without pivoting, and LU with partial pivoting. The automatic blocking of QR and pivoting LU is a notable achievement — to our knowledge, few previous compiler techniques, including theoretically more general loop transformation frameworks [21, 27, 1, 23, 31], were able to completely automate the blocking of these kernels, and none has produced the same blocking as produced by our technique. These results indicate that with low compilation cost, our technique can in practice match the effectiveness of much more expensive frameworks that are theoretically more powerful.

Key words: compiler optimization, loop transformation, complex loop structure, linear algebra kernels.

1 Introduction

Over the past twenty years, increases in processor speed have dramatically outstripped performance increases for standard memory chips, in both latency and bandwidth. To bridge this gap, architects typically insert multiple levels of cache between the processor and memory, resulting in a deep memory hierarchy. The data access latency to a higher level of the memory hierarchy is often orders of magnitude less than the latency to a lower level. To achieve high performance on such machines, compilers must optimize the locality of applications so that data fetched into caches are reused before being displaced.

Compiler researchers have developed many techniques to optimize the loop structures of applications for better locality. Of these techniques, the most widely used are a collection of unimodular and single loop transformations [32, 34, 8, 25, 7], such as loop blocking(tiling), reversal, skewing, interchange, distribution, fusion, and index-set splitting. These transformations are inexpensive and quite efficient in optimizing simple loop structures such as sequences of perfectly nested loops. However, on more complicated loop structures, these techniques often fail, even when an effective optimization is possible.

This paper extends these traditional transformation techniques to effectively optimize complex loop structures. We introduce a novel transformation, *dependence hoisting*, that facilitates the direct fusion and interchange of arbitrarily nested loops, and a new transformation framework that systematically combines *dependence hoisting* with other simple techniques to achieve aggressive loop interchange, blocking and fusion optimizations for better locality. Our framework is fast enough to be incorporated in most commercial production compilers.

We have implemented our framework as a Fortran source-to-source translator and have applied the translator to successfully optimize a collection of linear algebra kernels and real-word application benchmarks. The translator has broad applicability to general, non-perfectly nested loops. In particular, we emphasize applying it to block four numerical benchmark kernels: Cholesky, QR, LU factorization without pivoting, and LU factorization with partial pivoting. These kernels contain complex loop nests that are generally considered difficult to block automatically — to our knowledge, few previous compiler techniques have completely automated the blocking of QR and LU with pivoting. Our framework has successfully blocked all four benchmarks. The auto-blocked versions not only achieved significant performance improvements over the original unblocked versions, they also achieved a performance level comparable to that achieved by LAPACK, which was hand-optimized by expe-

```
do k = 1, n − 1
  do i = k + 1, n
s1:   a(i, k) = a(i, k)/a(k, k)
  enddo
  do j = k + 1, n
    do i = k + 1, n
s2:   a(i, j) = a(i, j)-a(i, k) ∗ a(k, j)
    enddo
  enddo
enddo
```

(a) KJI 

```
do i = 2, n
  do k = 1, i − 1
s1:   a(i, k) = a(i, k)/a(k, k)
    do j = k + 1, n
s2:   a(i, j) = a(i, j) − a(i, k) ∗ a(k, j)
    enddo
  enddo
enddo
```

(b) IKJ 

```
do j = 1, n
  do k = 1, j − 1
    do i = k + 1, n
s2:   a(i, j) = a(i, j) − a(i, k) ∗ a(k, j)
    enddo
  enddo
  do i = j + 1, n
s1:   a(i, j) = a(i, j)/a(j, j)
  enddo
enddo
```

(c) JKI 

Figure 1: Different versions of non-pivoting LU by Dongarra, Gustavson and Karp [13]

rienced algorithm designers using algorithm changes in some cases. These experimental results indicate that with a low compilation cost sufficient for even production compilers, our technique is quite powerful and can in practice match or exceed the real-world effectiveness (although not necessarily the theoretical effectiveness) of most existing general loop transformation frameworks [21, 27, 1, 23].

To illustrate the new loop transformation techniques, we use three equivalent versions of LU factorization without pivoting, as shown in Figure 1. Dongarra, Gustavson and Karp described various equivalent implementations of non-pivoting LU factorization [13], from which Figure 1 selects three implementations with different loop structures, each structure placing a different loop(k, i or j) at the outermost position. These chosen loop structures are particularly useful in blocking the non-pivoting LU code.

In Figure 1, the KJI version of non-pivoting LU in (a) is commonly used in scientific libraries such as the LINPACK collection [12]; both the IKJ form in (b) and JKI form in (c) are less commonly used versions with *deferred operations*: the IKJ version defers the scaling of each row of the matrix until immediately before the update of that row, and the JKI form defers the updates to each column until immediately before the scaling of that column. In (c), for example, at each iteration of the outermost j loop, statement s_2 first applies all the deferred updates to column j by subtracting multiples of columns 1 through $j - 1$; statement s_1 then scales column j immediately after these deferred updates.

Now, because all the loops(k, j and i) in Figure 1 carry data reuses, to achieve the best locality optimization, it is desirable to fully block the code by strip-mining all the loops and then shifting the strip-mined loops inside. In order to achieve this blocking effect, a compiler needs the ability to freely translate between each pair of the three versions in Figure 1; that is, it needs the ability to freely interchange the nesting order between any two of

the three loops(k,j and i).

The three code fragments in Figure 1 are not only equivalent, they also contain the same dependence constraints. A sophisticated compiler therefore should be able to recognize this equivalence and achieve all the translations. For example, to translate (b) to (a), a compiler can interchange the i and k loops in (b), distribute the interchanged i loop, and then interchange the distributed $i(s_2)$ loop (i loop surrounding s_2) with the $j(s_2)$ loop. A compiler can also easily reverse this procedure and translate (a) back to (b).

However, traditional unimodular loop transformation techniques cannot translate between Figure 1(a) (or (b)) and Figure 1(c) because these translations require the direct fusion of non-perfectly nested loops. For example, to translate from (a) to (c), a compiler needs to fuse the $k(s_1)$ loop (k loop surrounding s_1) with the $j(s_2)$ loop and then place the fused loop outside the $k(s_2)$ loop. However, the $k(s_1)$ loop encloses both statements s_1 and s_2 in the original code and therefore cannot be fused with loop $j(s_2)$ unless the outermost k loop is first distributed. Since a dependence cycle connecting s_1 and s_2 is carried by the outermost k loop, this k loop cannot be distributed before fusion. Lacking the ability to fuse the $k(s_1)$ and $j(s_2)$ loops when they are nested inside one another, traditional transformations cannot shuffle the nesting order of the these two loops and thus cannot establish the translation from (a) to (c).

To achieve the translation between (a) (or (b)) and (c), a compiler therefore needs the ability to directly fuse and interchange arbitrarily nested loops. The desired transformation should be able to automatically fuse a collection of non-perfectly nested loops and then shift the fused loop to the outermost position of the code segment containing the original loops. We have developed a novel technique, *dependence hoisting*,to achieve exactly the desired transformation. In addition, we have also extended the dependence model used in traditional unimodular loop transformation systems to automatically recognize the safety of applying *dependence hoisting* transformations. Our extended model includes a new dependence representation and a novel transitive dependence analysis algorithm that improves the average case efficiency of the previous Floyd-Warshall algorithm by Rosser and Pugh [29]. The extended model has a complexity comparable to that of the traditional dependence models and is described in more detail in Section 4.

We now briefly summarize the steps of applying *dependence hoisting* to translate Figure 1(a) to (c). After recognizing that it is legal to fuse the $k(s_1)$ and $j(s_2)$ loops in (a) at the outermost loop level, we first put a new

outermost dummy loop (with induction variable x) surrounding the original k loop in (a) (as shown in Figure 2). In the same step, we also insert conditionals surrounding s_1 and s_2 to synchronize the iterations of loops $j(s_2)$ and $k(s_1)$ with this new dummy loop. In particular, we insert conditionals to execute the iteration j of loop $j(s_2)$ only when $x = j$ and to execute the iteration k of loop $k(s_1)$ only when $x = k$. These conditionals shift the loop-carrying levels of the dependence cycles connecting s_1 and s_2 to the new outermost x loop. Because the original k loop is now inside the x loop and no longer carries the dependence cycles connecting s_1 and s_2, we can distribute this k loop and translate (a) to (c). Section 2 describes the transformation in more detail.

The transformation technique in this paper is similar to a technique called *iteration space slicing* proposed by Pugh and Rosser [31] in that they also apply fusion to related iterations of statements without being limited by the original nesting structure of the loops surrounding these statements. Pugh and Rosser manipulate the iteration spaces or data spaces of statements to summarize related computations. In contrast, we focus on transforming loops directly and do not manipulate any symbolic spaces of statements. Although our approach is less general than the approach by Pugh and Rosser, it is powerful enough for a large class of real-world applications and is much more efficient.

Besides iteration space slicing, several other general loop transformation frameworks [21, 27, 1, 23] can also aggressively transform complicated loop structures. These frameworks typically adopt a mathematical formulation of program dependences and loop transformations. They first map the iteration spaces of statements into some unified space, such as the time space or data space of a program. The unified space is then considered for transformation. Finally, a new program is constructed by mapping the selected transformations of the unified space onto the iteration spaces of statements. These frameworks are powerful because they incorporate the whole solution space of program transformations. However, they are also complicated and expensive to apply. Because of their high cost, these frameworks are rarely used in commercial compilers.

Our strategy is a compromise that trades a small amount of generality for substantial gains of efficiency. In particular, we do not manipulate any mathematically formulated space other than the iteration space of loops and the dependences within that space. Our technique can be integrated into traditional unimodular transformation systems and, because it is inexpensive, it should be suitable for inclusion in commercial production compilers.

The remainder of this paper is organized as follows. Section 2 first illustrates the dependence hoisting

transformation by translating the KJI form of non-pivoting LU in Figure 1(a) to the JKI form in (c). Section 3 then presents some notations and definitions for further elaboration of this paper. Section 4 presents an extended dependence model to determine the safety of transforming arbitrarily nested loops. Section 5 then presents the dependence hoisting transformation algorithms in detail. Section 6 presents a framework that systematically applies dependence hoisting for better locality. Section 7 presents experimental results. Section 8 introduces related work. Finally, conclusions are drawn in Section 9.

2 Example: Translating Non-pivoting LU

This section illustrates how to translate the KJI form of non-pivoting LU in Figure 1(a) into the JKI form in (c). As discussed in Section 1, this translation requires fusing the $k(s_1)$ and $j(s_2)$ loops in (a) and then shift the fused loop to the outermost loop level. To facilitate this transformation, the $k(s_1, s_2)$ loop needs to be distributed. However, due to the dependence cycle connecting s_1 and s_2, the distribution is not legal in the original code.

We introduce a novel transformation called *dependence hoisting* to resolve this conflict. A dependence hoisting transformation fuses a collection of arbitrarily nested loops and then places the fused loop at the outermost position of a code segment containing the original loops. It therefore permits the direct fusion and interchange of non-perfect loop nests and is particularly useful when the collection of loops cannot be separated due to dependence cycles, as is the case of fusing loops $k(s_1)$ and $j(s_2)$ (or interchanging loops $k(s_2)$ and $j(s_2)$) in Figure 1(a).

Figure 2(a) shows the original KJI form of non-pivoting LU along with the dependence information between statements s_1 and s_2, where each dependence edge is marked with dependence relations between iterations of the loops surrounding these statements. Note that these relations involve not only iterations of common loops surrounding both s_1 and s_2 (for example, $k(s_1, s_2)$), but also non-common loops surrounding only one of the statements (for example, $j(s_2)$). The extra information is necessary to precisely model dependences independent of the original loop structure. The extended dependence representation is described in more detail in Section 4.1.

We translate the KJI form in Figure 2(a) into the JKI form in Figure 1(c) in three steps. First, we create

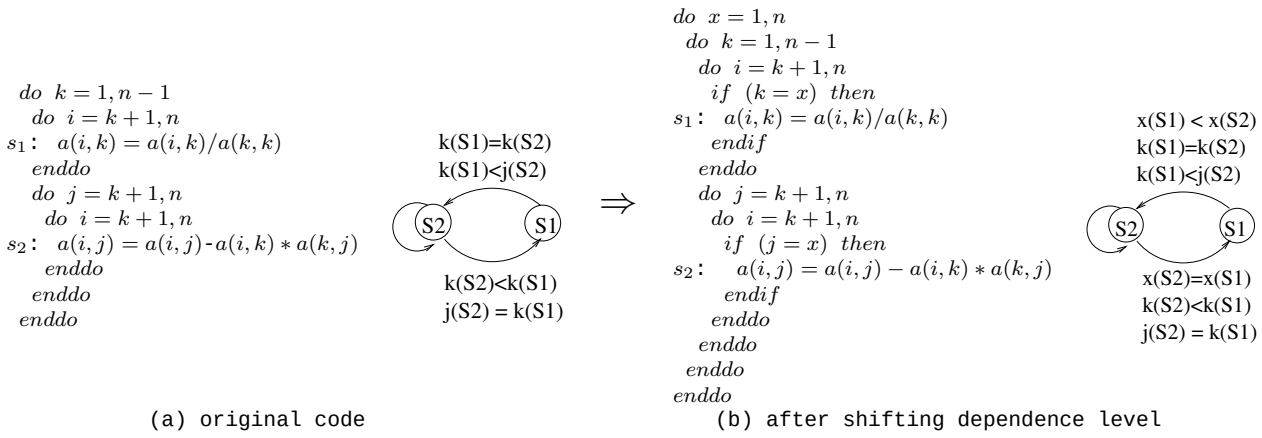
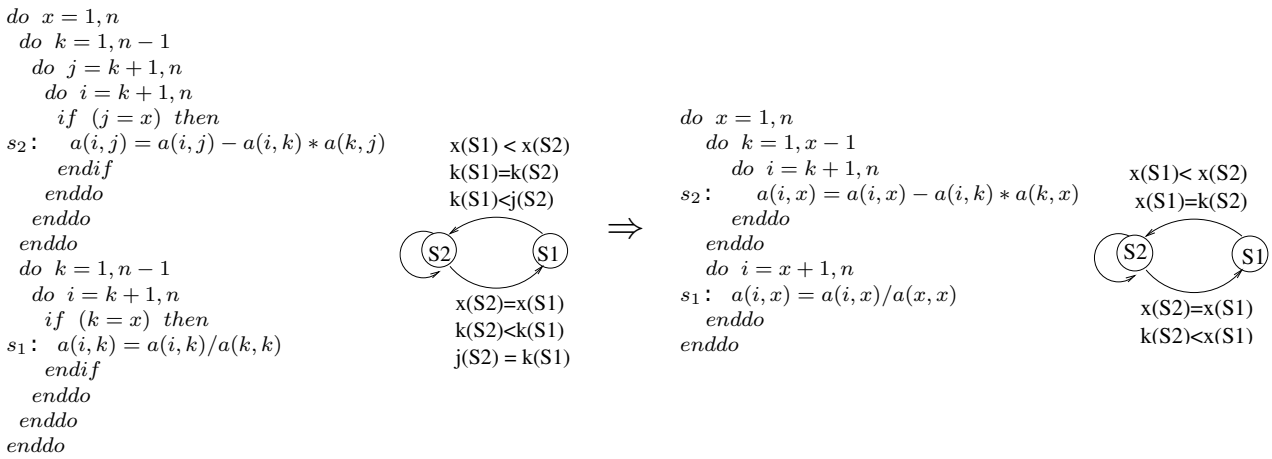


Figure 2: Transforming KJI version of non-pivoting LU. Step(1): shift dependence levels

a new dummy loop surrounding the original code in Figure 2(a). This dummy loop has an index variable x that iterates over the union of the iteration ranges of loops $k(s_1)$ and $j(s_2)$. In the same step, we insert conditionals in (a) so that statement s_1 is executed only when $x = j$ and statement s_2 is executed only when $x = k$. Figure 2(b) shows the result of this step, along with the modified dependence edges which include dependence relations involving iterations of the new outermost x loop.

Now, because the conditionals $x = k$ and $x = j$ in Figure 2(b) synchronize the $k(s_1)$ and $j(s_2)$ loops with the new $x(s_1, s_2)$ loop in a lock-step fashion, loop $x(s_1)$ always has the same dependence conditions as those of loop $k(s_1)$, and loop $x(s_2)$ always has the same dependence conditions as those of loop $j(s_2)$. As shown in the dependence graph of (b), the new outermost x loop now carries the dependence edge from s_1 to s_2 and thus carries the dependence cycle connecting s_1 and s_2. This shifting of dependence level makes it possible for the second transformation step to distribute the $k(s_1, s_2)$ loop in (b), which no longer carries a dependence cycle. The transformed code after distribution is shown in Figure 3(a). Note that this step requires interchanging the order of statements s_1 and s_2.

Finally, we can now remove the redundant loops $k(s_1)$ and $j(s_2)$ in Figure 3(a) along with the conditionals that synchronize them with the outermost x loop. To legally remove these loops and conditionals, we substitute the index variable x for the index variables of the removed loops $k(s_1)$ and $j(s_2)$. In addition, we adjust the upper bound of the $k(s_2)$ loop to $x - 1$, in effect because the $j(s_2)$ loop is exchanged outward before being removed. The transformed code after this cleanup step is shown in Figure 3(b).



(a) after distributing $k(s1, s2)$ (b) after cleanup

Figure 3: Transforming KJI version of non-pivoting LU. Steps (2) and (3): distribute loops and cleanup

Notations	Definitions
$\ell(s)$	loop ℓ surrounding statement s
$Ivar(\ell_i(s))$	the iteration index variable of loop $\ell_i(s)$
$Range(\ell_i(s))$	the iteration range of loop $\ell_i(s)$
$I : \ell_i(s)$	iteration I of loop $\ell_i(s)$
$d(s_x, s_y)$	the set of dependence edges from statement s_x to s_y in the dependence graph
$td(s_x, s_y)$	the set of transitive dependence edges from s_x to s_y in the trasitive dependence graph

Table 1: Notations for loops and dependences

The final transformed code in Figure 3(b) is the same as the JKI form of non-pivoting LU in Figure 1(c) except that the name of the outermost loop index variable is x instead of j. In reality, the index variables of the new loops can often reuse those of the removed loops so that a compiler does not have to create a new loop index variable at each dependence hoisting transformation.

3 Notations and Definitions

This section defines notations used throughout the elaboration of this paper. These notations, summarized in Table 1, include the modeling of loop and statement iterations, the treatment of conditionals in loop transformations, and the notations for program dependences.

To transform loops without being limited by their original nesting structure, we adopt a loop model similar to that used by Ahmed, Mateev and Pingali [1]. We use the notation $\ell(s)$ to denote a loop ℓ surrounding some statement s and thus treat loop $\ell(s)$ as different from loop $\ell(s')$ ($s' \neq s$). This strategy effectively treats each loop

ℓ surrounding multiple statements as potentially distributable. Dependence analysis will be used to determine whether or not each loop can actually be distributed.

Each statement s inside a loop is executed multiple times; each execution is called an *iteration instance* of s. Suppose that statement s is surrounded by m loops $\ell_1, \ell_2, ..., \ell_m$. For each loop $\ell_i(s)(i = 1, ..., m)$, the iteration index variable of ℓ_i is denoted as $Ivar(\ell_i(s))$, and the iteration range is denoted as $Range(\ell_i(s))$. Each value I of $Ivar(\ell_i(s))$ defines a set of iteration instances of statement s, a set that can be expressed as $Range(\ell_1) \times ... \times Range(\ell_{i-1}) \times I \times ... \times Range(\ell_m)$. This iteration set is denoted as *iteration* $I : \ell_i(s)$ or *the iteration* I *of loop* $\ell_i(s)$.

In order to focus on transforming loops, we treat all the other control structures in a program as primitive statements; that is, we do not transform any loops inside other control structures such as conditional branches. This strategy is adopted to simplify the technical presentation of this paper. To optimize real-world applications, preparative transformations such as "if conversion" [3] must be incorporated to remove the non-loop control structures in between loops. These preparative transformations are outside the scope of this paper and will not be discussed further.

In our extended program dependence model, we use both dependences and transitive dependences between iteration instances of statements to model the safety of loop transformations. The set of transitive dependences from statement s_x to s_y summarizes the complete dependence relations from s_x to s_y and is computed by performing transitive analysis on the dependence graph. Given a dependence graph, we can hence construct a transitive dependence graph accordingly, where the set of transitive dependence edges from statement s_x to s_y summarizes all the dependence paths from s_x to s_y in the dependence graph.

Throughout this paper, we use the notation $d(s_x, s_y)$ to denote the set of dependence edges from a statement s_x to s_y in the dependence graph, and use the notation $td(s_x, s_y)$ to denote the set of transitive dependence edges from s_x to s_y. Each transitive dependence edge in $td(s_x, s_y)$ models a dependence path from s_x to s_y and has the exact same representation as that of each dependence edge in $d(s_x, s_y)$, as described in Section 4.1.

4 Extended Dependence Model

This section presents our dependence model for determining the safety of transforming arbitrarily nested loops. To achieve this, we make two changes to the traditional dependence model used in most unimodular loop transformation systems. First, we associate each dependence edge with a new representation, *extended direction matrix* (EDM), that specifies not only dependence relations between iteration numbers of common loops, but also relations between non-common loops. Second, we apply transitive analysis on the dependence graph to summarize the complete dependence information between statements. These summarized transitive dependences are then used to determine the safety of transforming loops independent of their original nesting structure.

The EDM dependence representation in this paper is less powerful but also much less expensive than those adopted by previous more general transformation frameworks [21, 27, 1, 23, 31], which use integer set mappings [16] to precisely represent the dependence relations between iterations of statements. The integer set mapping representations are quite expensive and can incur exponential cost when used in summarizing transitive dependences. Although our dependence representation is less precise than the symbolic integer set mappings, it is sufficient for optimizing a large class of real-world applications and has a much lower cost.

To effectively summarize the transitive dependence relations between statements, we also developed a demand-driven transitive analysis algorithm that efficiently summarizes all the dependence paths to a single destination or from a single source vertex. Because transitive dependence analysis needs to summarize all the paths between two statements, it is a path summary problem rather than a simple reachability problem on directed graphs. When summarizing dependence paths between all pairs of statements, our algorithm has the same worst case complexity ($O(N^3)$ for N statements) as the previous enhanced Floyd-Warshall algorithm by Rosser and Pugh [29]. However, because Rosser and Pugh's algorithm need to solve the all-pairs path summary problem up front, their algorithm has $O(N^3)$ complexity for all graphs with N vertices. Our algorithm performs much better in the average case and can compute single destination (or source) vertex path summaries in linear time for most of the dependence graphs encountered in practice.

Note that we apply transitive dependence analysis only when transforming non-perfectly nested loops. To determine the safety of transforming perfect loop nests, we use the traditional dependence analysis based on individual dependence edges. This strategy further reduces the cost of incorporating transitive dependence

analysis in a compiler. In practice, transitive dependence analysis proves to incur a similar cost as that incurred by traditional dependence analysis in our compiler (see Section 7.3). It is therefore efficient enough to be incorporated into production compilers.

The following elaborates both the EDM dependence representation and the new transitive dependence analysis algorithm. Section 4.1 introduces the EDM representation of dependences. Section 4.2 defines a set of operations on the EDM representation and on sets of EDMs. Building on these operations, Section 4.3 presents the transitive dependence analysis algorithm, which was also earlier published in [36].

4.1 Dependence Representation

This section introduces a new dependence representation, *Extended Direction Matrix (EDM)*, to model the dependence conditions between arbitrarily nested loops surrounding two statements. Suppose that statements s_x and s_y are surrounded by m_x loops (ℓ_{x1}, ℓ_{x2}, $\ldots \ell_{xm_x}$) and m_y loops (ℓ_{y1}, ℓ_{y2}, $\ldots \ell_{ym_y}$) respectively. A dependence EDM from s_x to s_y is an $m_x \times m_y$ matrix D. Each entry $D[i,j]$ ($1 \leq i \leq m_x$, $1 \leq j \leq m_y$) in the matrix specifies a relation between iterations of loops $\ell_{xi}(s_x)$ and $\ell_{yj}(s_y)$. The dependence represented by D satisfies the conjunction of all these conditions.

For each dependence EDM D from statement s_x to s_y, the dependence condition between loops $\ell_{xi}(s_x)$ and $\ell_{yj}(s_y)$ is denoted as $D(\ell_{xi}, \ell_{yj})$. Each condition $D(\ell_{xi}, \ell_{yj})$ can have the following values: "$= n$", "$\leq n$", "$\geq n$" and "$*$", where n is a small integer called an *alignment factor*. The first three values "$= n$", "$\leq n$" and "$\geq n$" specify that the dependence conditions are $Ivar(\ell_{xi}) = Ivar(\ell_{yj}) + n$, $Ivar(\ell_{xi}) \leq Ivar(\ell_{yj}) + n$ and $Ivar(\ell_{xi}) \geq Ivar(\ell_{yj}) + n$ respectively; the last value "$*$" specifies that the dependence condition is always true. The *dependence direction* ("=", "$\leq$","$\geq$" or "$*$") of the condition is denoted as $Dir(D(\ell_{xi}, \ell_{yj}))$ and the *alignment factor* of the condition is denoted as $Align(D(\ell_{xi}, \ell_{yj}))$.

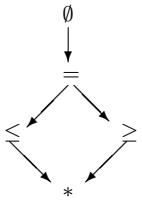
A dependence EDM extends the traditional dependence vector representation [3, 35] by computing a relation between iterations of two loops $\ell_x(s_x)$ and $\ell_y(s_y)$ even if $\ell_x \neq \ell_y$. The extra information can be computed using traditional dependence analysis techniques [3, 35, 5], which are well-understood and will not be further discussed in this paper.

$* \cdot x \rightarrow * \; \forall x$
$= n_1 \cdot = n_2 \rightarrow = (n_1 + n_2)$
$= n_1 \cdot \leq n_2 \rightarrow \leq (n_1 + n_2)$
$= n_1 \cdot \geq n_2 \rightarrow \geq (n_1 + n_2)$
$\leq n_1 \cdot \leq n_2 \rightarrow \leq (n_1 + n_2)$
$\geq n_1 \cdot \geq n_2 \rightarrow \geq (n_1 + n_2)$
$\geq n_1 \cdot \leq n_2 \rightarrow *$

(a) concatenation

$* \wedge x \rightarrow x \; \forall x$
$= n_1 \wedge = n_2 \rightarrow = n_1$ *if* $n_1 = n_2$; $\emptyset$ *otherwise*
$\leq n_1 \wedge = n_2 \rightarrow = n_2$ *if* $n_2 \leq n_1$; $\emptyset$ *otherwise*
$\geq n_1 \wedge = n_2 \rightarrow = n_2$ *if* $n_2 \geq n_1$; $\emptyset$ *otherwise*
$\geq n_1 \wedge \leq n_2 \rightarrow = \{n_1, ..., n_2\}$ *if* $n_1 \leq n_2$; $\emptyset$ *otherwise*
$\leq n_1 \wedge \leq n_2 \rightarrow \leq min(n_1, n_2)$
$\geq n_1 \wedge \geq n_2 \rightarrow \geq max(n_1, n_2)$

(b) conjunction



(c) direction lattice

Figure 4: Operations on dependence conditions

4.2 Operations on Dependence EDMs

This section defines a set of operations on the dependence EDM representation. These operations are required by the transitive dependence analysis algorithm presented in Section 4.3. In the following, Section 4.2.1 defines operations on individual dependence EDMs, and Section 4.2.2 defines operations on sets of EDMs.

4.2.1 Operations on Individual EDMs

This section defines two operators, *concatenation*$(\cdot)$ and comparison $(\leq)$, on individual dependence EDMs. Because each EDM models a path in the dependence graph, these operators can be used to summarize different paths in the dependence graph.

The first operator, concatenation$(\cdot)$, is applied to two EDMs, D_{xy} and D_{yz}, where D_{xy} represents a dependence path p_1 from statement s_x to s_y, and D_{yz} represents a path p_2 from statement s_y to s_z. The concatenation of D_{xy} and D_{yz} $(D_{xy} \cdot D_{yz})$ represents the dependence path p_1p_2 along $s_x \rightarrow s_y \rightarrow s_z$. Suppose that statements s_x,s_y and s_z are surrounded by m_x, m_y and m_z loops respectively. The *concatenation* of D_{xy} ($m_x \times m_y$ matrix) and D_{yz}($m_y \times m_z$ matrix) is an $m_x \times m_z$ matrix D_{xz} satisfying

$$D_{xz}[i,j] = \bigwedge_{1 \leq k \leq m_y} (D_{xy}[i,k] \cdot D_{yz}[k,j]), \tag{1}$$

where the concatenation$(\cdot)$ and conjunction $(\wedge)$ operators of dependence conditions are defined in Figure 4.

In Equation (1), the concatenation of dependence conditions $D_{xy}[i,k]$ and $D_{yz}[k,j]$ $(D_{xy}[i,k] \cdot D_{yz}[k,j])$ computes the dependence condition for the path $\ell_{xi} \rightarrow \ell_{yk} \rightarrow \ell_{zj}$, where ℓ_{xi}, ℓ_{yk} and ℓ_{zj} are the ith, kth and jth loops surrounding statements s_x, s_y and s_z respectively. For example, suppose $D_{xy}[i,k] =$ " ≤ -1" and $D_{yz}[k,j] =$ " ≤ 0". Given two arbitrary iterations $I : \ell_{xi}$ (iteration I of loop ℓ_{xi}) and $J : \ell_{zj}$, the dependence from I to J exists only if there is an iteration $K : \ell_{yk}$ that is also involved in the dependence path; that is, the

dependence exists only if iterations $I : \ell_{xi}$ and $K : \ell_{yk}$ satisfy condition $D_{xy}[i, j]$ (that is, $I \leq K - 1$) and if iterations $K : \ell_{yk}$ and $J : \ell_{zj}$ satisfy condition $D_{yz}[k, j]$ (that is, $K \leq J$). Thus the relation $I \leq K - 1 \leq J - 1$ must holds; that is, $D_{xy}[i, k] \cdot D_{yz}[k, j] =$ " ≤ -1". Note that the dependence condition between loops ℓ_{xi} and ℓ_{zj} must involve at least one iteration of each loop surrounding statement s_y; otherwise, the involved iteration set for s_y is empty, indicating that the dependence path does not exist. Therefore each dependence condition $D_{xz}[i, j]$ in Equation (1) must conjoin the dependence conditions for every path $\ell_{xi} \rightarrow \ell_{yk} \rightarrow \ell_{zj}$, where ℓ_{yk} is a loop surrounding statement s_y.

The second operator, comparison ($\leq$), of dependence EDMs is applied to two EDMs, D_1 and D_2, that both represent dependence paths from statement s_x to s_y. Suppose that s_x and s_y are surrounded by m_x loops and m_y loops respectively. The following equation defines the comparison operator on D_1 and D_2:

$$D_1 \leq D_2 \Leftrightarrow D_1[i, j] \leq D_2[i, j] \quad \forall 1 \leq i \leq m, 1 \leq j \leq n \tag{2}$$

Here the comparison ($\leq$) operator for the dependence conditions $D_1[i, j]$ and $D_2[i, j]$ is defined as

$$D_1[i, j] \leq D_2[i, j] \Leftrightarrow D_1[i, j] \wedge D_2[i, j] = D_1[i, j], \tag{3}$$

and the conjunction ($\wedge$) operator on dependence conditions is defined in Figure 4(b). If $D_1[i, j] \wedge D_2[i, j] = D_1[i, j]$, the dependence relation defined by $D_1[i, j]$ is part of that defined by $D_2[i, j]$; that is, the loop iterations satisfying the condition $D_1[i, j]$ is a subset of those satisfying $D_2[i, j]$. Equation (2) thus specifies that EDM D_1 is subsumed by D_2 (that is, $D_1 \leq D_2$) only if for every pair of loops, $\ell_i(s_x)$ and $\ell_j(s_y)$, the dependence condition $D_1(\ell_i, \ell_j)$ is subsumed by the condition $D_2(\ell_i, \ell_j)$ (that is, $D_1(\ell_i, \ell_j) \leq D_2(\ell_i, \ell_j)$). Because a dependence path represented by D_1 must satisfy all the conditions in D_1 (similarly for D_2), if $D_1 \leq D_2$, the dependence conditions modeled by D_1 are a subset of those modeled by D_2, in which case the dependence path represented by D_1 is redundant (subsumed by D_2) and can be ignored.

The comparison operators on dependence conditions and EDMs, as defined by Equations (3) and (2), organize the dependence conditions and EDMs into lattices. In particular, since there are only four different dependence directions ($=$, $\leq$, $\geq$ and $*$), these directions form a lattice of depth 3, shown in Figure 4(c). However, the lattice of dependence conditions has infinite depth because there are infinite numbers of different integer alignment factors. As a result, the lattice of dependence EDMs also becomes infinite.

To limit the total number of different dependence conditions and EDMs, we restrict the absolute magnitude of dependence alignment factors to be less than a small constant (for example, 5) and use approximations for conditions with alignment factors of magnitudes larger than the threshold. Because the depth of a loop nest is typically bounded by a small constant, by limiting the total number of different dependence conditions, we also limit the total number of different EDMs to be a constant, making it possible for most operations on EDMs to finish within some constant time. In practice, this strategy significantly improves the efficiency of summarizing dependence paths. And because most loops are in reality "misaligned" with each other by only 1-2 iterations, this restriction rarely sacrifices the generality of loop transformations.

4.2.2 Operations on Dependence EDM Sets

This section defines three operators, union($\cup$), concatenation($\cdot$) and transitive closure($*$), on sets of dependence EDMs. Each set $td(s_x, s_y)$ contains one or more dependence paths between the same source and sink statements. Thus all the EDMs in $td(s_x, s_y)$ have the same row and column dimensions.

The first operator, union($\cup$), is applied to two EDM sets, $td_1(s_x, s_y)$ and $td_2(s_x, s_y)$, both of which represent dependence paths from statement s_x to s_y. The set-union of $td_1(s_x, s_y)$ and $td_2(s_x, s_y)$ thus is defined as

$$td_1(s_x, s_y) \cup td_2(s_x, s_y) = \{D \mid D \in td_1(s_x, s_y) \text{ or } D \in td_2(s_x, s_y)\}. \tag{4}$$

Note that redundant EDMs are removed from the above set-union result; that is, if two EDMs D_1 and D_2 both belong to $td_1 \cup td_2$, and if D_1 is subsumed by D_2 ($D_1 \leq D_2$ as defined by Equation (2)), D_1 is redundant and is removed from $td_1 \cup td_2$. In practice, each set usually contains only a few EDMs after removing redundant paths. Most operations on EDM sets thus can finish within some small constant time limit.

The second operator, concatenation ($\cdot$), is applied to two EDM sets, $td(s_x, s_y)$ and $td(s_y, s_z)$, where $td(s_x, s_y)$ represents dependence paths from statement s_x to s_y, and $td(s_y, s_z)$ represents paths from s_y to s_z. The concatenation of $td(s_x, s_y)$ and $td(s_y, s_z)$ summarizes all dependence paths that start from s_x to s_y following a path in $td(s_x, s_y)$ and then continue from s_y to s_z following a path in $td(s_y, s_z)$; that is, it summarizes the paths from statement s_x to s_z, passing through statement s_y. The concatenation of $td(s_x, s_y)$ and $td(s_y, s_z)$ is defined as

$$td(s_x, s_y) \cdot td(s_y, s_z) = \{D_{xy} \cdot D_{yz} \mid D_{xy} \in td(s_x, s_y) \text{ and } D_{yz} \in td(s_y, s_z)\}, \tag{5}$$

where the concatenation of individual paths ($D_{xy} \cdot D_{yz}$) is defined by Equation (1).

The third operator, transitive closure ($*$), is applied to a single EDM set, $td(s, s)$, that represents dependence paths from a single statement s to itself. This operator summarizes the cycles from s to itself and is defined as

$$td(s,s)^* = td(s,s) \cup (td(s,s) \cdot td(s,s)) \cup (td(s,s) \cdot td(s,s) \cdot td(s,s)) \cup \ldots \quad (6)$$

Here the infinite unions and concatenations stop when a fixed point is reached. Because the result of $td(s, s)^*$ is simplified at each union operation of the EDM sets, and because there are only a limited number of different EDMs (see Section 4.2.1), each transitive closure operation is guaranteed to reach a fixed point and terminate after a constant number of set unions and concatenations.

4.3 Transitive Dependence Analysis Algorithm

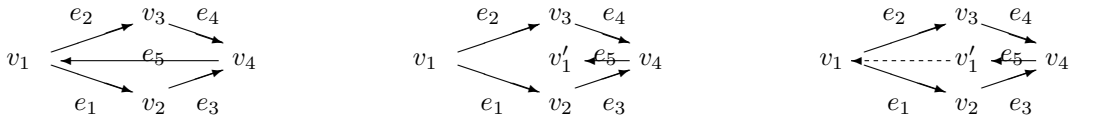
This section introduces a new transitive dependence analysis algorithm to summarize on demand the complete dependence information between statements. This algorithm is independent of specific dependence representations. In particular, the algorithm can be applied using the EDM representation of dependence paths described in Section 4.1.

Figure 6 shows the transitive analysis algorithm, which has three steps. The first step transforms an arbitrary dependence graph into an acyclic graph (a DAG) by splitting a set of vertices. The second step summarizes the acyclic paths in the DAG. Finally, the third step extends the acyclic path summaries to include cycles in the original graph. Section 4.3.1 uses an example to illustrate these steps. Sections 4.3.2, 4.3.3, and 4.3.4 then describe the three steps respectively. Finally, the complexity of the entire algorithm is discussed in Section 4.3.5.

4.3.1 A Simple Example

This section illustrates the transitive dependence analysis algorithm in Figure 6 using a simple dependence graph shown in Figure 5(a). To summarize transitive dependences for this graph, the algorithm performs the following steps.

First, to break the dependence cycles in (a), the algorithm visits all the vertices in the graph in depth-first order and identifies all the *back edges* that go back to already visited vertices. The algorithm then breaks the dependence cycles by modifying these back edges. Suppose that the vertex v_1 in (a) is visited first and that



(a) original graph (b) after splitting vertices (c) after recovering cycles

Figure 5: Example for transitive dependence analysis algorithm

edge e_5 is identified as a back edge. The algorithm splits vertex v_1 with a twin vertex v_1' and then changes edge e_5 to go into v_1' instead. The transformed graph is shown in (b). Here because v_1' has no outgoing edges, the transformed graph no longer has dependence cycles.

The algorithm then pre-computes the cycle information (denoted as $C(v_1, v_1')$) for vertex v_1 so that the broken cycle at v_1 can be recovered later. To compute $C(v_1, v_1')$, the algorithm first traverses each vertex in the transformed DAG in (b) in reverse topological order. When visiting each vertex v, the algorithm computes the single-destination path summary from v to v_1'. The computed path summaries include $td(v_4, v_1') = \{EDM(e_5)\}$, $td(v_3, v_1') = EDM(e_4) \cdot td(v_4, v_1')$, $td(v_2, v_1') = EDM(e_3) \cdot td(v_4, v_1')$, and $td(v_1, v_1') = (EDM(e_1) \cdot td(v_2, v_1')) \cup (EDM(e_2) \cdot td(v_3, v_1'))$. The cycle information $C(v_1, v_1')$ is then set to be $td(v_1, v_1')^*$, the transitive closure of $td(v_1, v_1')$.

The algorithm is now ready to compute path summaries for the original dependence graph in (a). This graph is equivalent to the graph shown in (c), which has inserted a new *dummy* edge from vertex v_1' to v_1. All the cycle information can be recovered by reconnecting this dummy edge that has been broken in (b). For example, to compute path summary from v_3 to v_4 in the original graph, the algorithm first computes $td(v_3, v_4)$ on the transformed DAG in (b), which yields $td(v_3, v_4) = \{EDM(e_4)\}$. To recover the dummy edge $v_1' \rightarrow v_1$, the algorithm recomputes $td(v_3, v_4)$ as $\{EDM(e_4)\} \cup (td(v_3, v_1') \cdot C(v_1, v_1') \cdot td(v_1, v_4))$, where $td(v_3, v_1') = EDM(e_4) \cdot EDM(e_5)$, $C(v_1, v_1') = ((EDM(e_1) \cdot EDM(e_3) \cdot EDM(e_5)) \cup (EDM(e_2) \cdot EDM(e_4) \cdot EDM(e_5)))^*$, and $td(v_1, v_4) = EDM(e_2) \cdot EDM(e_4)$. The details of these steps are explained in more detail in the following sections.

4.3.2 Preprocessing Cycles

This section describes the first step of the transitive dependence analysis algorithm, a step encoded by the function *Preprocess-Cycles* in Figure 6. This step transforms an arbitrary dependence graph into a directed

Trans-Dep-Anal(G, v)
  G: dependence graph; v: destination vertex
if (G has not been preprocessed) then
  Preprocess-Cycles(G)
Path-Summary-On-Cyclic-Graphs(G, v)

Preprocess-Cycles(G)
Tarjan-SCC($G, SCCs, BackEdges$)
for (each edge $e : p \rightarrow v$ in $BackEdges$) do
  v' = Split-vertex(v); Change e to go to v'
for each $scc \in SCCs$ with split vertices $(v_1, v_1'), ...(v_m, v_m')$
  for $i = 1, m$ do
    Path-Summary-On-DAG(scc, v_i')
    for $k = 1, i-1$ do
      for each vertex $p \in scc$ do
        $td(p, v_i') = td(p, v_i') \cup td(p, v_k') \cdot C(v_k, v_k') \cdot td(v_k, v_i')$
    $C(v_i, v_i') = td(v_i, v_i')^*$

Path-Summary-On-DAG(G, v)
  G: acyclic dependence graph
  v: destination vertex
$td(v, v)$ = { identity EDM of statement v }
for each statement p in G in reverse topological order
  $td(p, v) = \emptyset$
  for each edge $e : p \rightarrow q$ in G do
    $td(p, v) = td(p, v) \cup \{EDM(e)\} \cdot td(q, v)$

Path-Summary-On-Cyclic-Graphs(G, v)
  G: dependence graph
  v: destination vertex
for each $scc \in SCCs(G)$ in reverse topological order do
  Path-Summary-On-DAG(G, v)
  for (each split vertex (v_k, v_k') in scc) do
    for (each vertex $p \in scc$) do
      $td(p, v) = td(p, v) \cup td(p, v_k') \cdot C(v_k, v_k') \cdot td(v_k, v)$

Figure 6: Transitive dependence analysis algorithm

acyclic graph (a DAG) and then pre-computes information so that the broken cycles may be recovered later.

To break all the cycles in the original graph G, the algorithm first uses the well-known Tarjan SCC algorithm (function *Tarjan-SCC* in Figure 6) to find all the strongly connected components (SCCs) in G. The Tarjan SCC algorithm uses a stack to track the traversal of vertices in G and by doing so, identifies all the *back edges* that go to already visited vertices in each SCC. The transitive analysis algorithm then translates each SCC into a DAG by reconnecting these back edges. For each vertex v_i that has an incoming back edge $p \rightarrow v_i$, the algorithm splits v_i with a twin vertex v_i'. It then changes all the back edges into v_i to go to v_i' instead. Because all the back edges now go to the new twin vertices which have no outgoing edges, all the original cycles are broken. The transformed dependence graph thus becomes a DAG.

After transforming the original dependence graph into a DAG, the algorithm then pre-computes information so that the broken cycles can be recovered later. For each strongly connected component in the original graph, suppose that m vertices are split and that the twin vertices are $(v_1, v_1'), \ldots (v_m, v_m')$. For each pair of twin vertices v_i and v_i' $(1 \leq i \leq m)$, the following cycle information is pre-computed:

- $td(p, v_i')$ $\forall$ vertex p in SCC: cycle summary from p to v_i'. The summary includes all the original cycles involving vertices $v_1, \ldots v_{i-1}$.
- $C(v_i, v_i') = (td(v_i, v_i'))^*$: cycle summary from v_i to itself. The summary includes all the original cycles involving vertices $v_1, \ldots v_i$.

Section 4.3.4 describes how to use the above information to recover broken cycles.

To pre-compute the cycle information for each split vertex v_i, the algorithm first computes the single-

destination path summaries to v'_i on the transformed DAG (described in Section 4.3.3). It then extends each cycle summary $td(p, v'_i)$ to include all paths $p \rightsquigarrow v'_k \rightarrow v_k \rightsquigarrow \ldots \rightsquigarrow v'_k \rightarrow v_k \rightsquigarrow v'_i \; \forall 1 \leq k < i$. Here $v'_k \rightarrow v_k$ is conceptually a dummy edge from v'_k to v_k, as shown in Figure 5(c), and $v_k \rightsquigarrow \ldots \rightsquigarrow v'_k$ includes all the cycles involving $v_1, \ldots v_k$, the cycles that have already been computed correctly.

4.3.3 Path Summary On DAG

The section describes the second step of the transitive dependence analysis algorithm, a step encoded by the function *Path-Summary-On-DAG* in Figure 6. This function summarizes dependence paths into a single destination vertex v on a directed acyclic graph G. The algorithm has time complexity linear to the size of the graph.

To compute $td(p, v)$ for each vertex $p \in G$, the algorithm first initiates the self transitive dependence of v ($td(v, v)$) using an identity EDM. Suppose v is surrounded by m loops, the identity EDM for v is a $m \times m$ matrix D_{mm} that satisfies the following condition:

$$DI[i, j] = \text{“}= 0\text{”} \; if \; i = j; \; \text{“}*\text{”} \; if \; i \neq j, \forall 1 \leq i, j \leq m \tag{7}$$

The above definition for DI guarantees that $DI \cdot D_1 = D_1$ for each $m * n$ EDM D_1 and $D_2 \cdot DI = D_2$ for each $n * m$ EDM D_2.

The algorithm then computes $td(p, v)$ for each vertex p in the reverse topological order of the DAG. For each edge $e : p \rightarrow q$ in the dependence DAG, $EDM(e)$ denotes the EDM associated with e. Because the vertices in the DAG are traversed in reverse topological order, for each edge $e : p \rightarrow q$, $td(q, v)$ has already been computed correctly. The algorithm thus computes $td(p, v)$ as the union of $EDM(e) \cdot td(q, v)$ for each edge $e : p \rightarrow q$ that leaves vertex p.

4.3.4 Path Summary on Cyclic Graphs

This section describes the final step of the transitive dependence analysis algorithm in Figure 6, a step encoded by the function *Path-Summary-On-Cyclic-Graphs*. This function summarizes the dependence paths to a single destination vertex v on an arbitrary dependence graph G.

To compute the path summaries from all vertices in the dependence graph G to a particular vertex v, the algorithm computes path summaries for a single SCC(strongly connected component) of G at a time. The algorithm traverses the SCCs of G in reverse topological order, which guarantees that when summarizing paths for each SCC, all the information for the SCCs closer to v have already been computed correctly.

The algorithm computes path summaries for each SCC in two steps. First, it computes all the path summaries on the transformed DAG of the SCC. It then extends each computed path summary $td(p, v)$ with the recovered broken cycles. For each split twin vertex pair (v_k, v'_k) of the SCC, the dummy edge from v'_k to v_k is recovered for $td(p, v)$ using the following equation:

$$td(p, v) = td(p, v) \cup td(p, v'_k) \cdot C(v_k, v'_k) \cdot td(v_k, v) \tag{8}$$

This equation extends $td(p, v)$ with all paths $p \rightsquigarrow v'_k \rightsquigarrow \ldots \rightsquigarrow v_k \rightsquigarrow v$, where the paths $v'_k \rightsquigarrow \ldots \rightsquigarrow v_k$ includes all the cycles involving $v_1, \ldots v_k$, and the paths $p \rightsquigarrow v'_k$ and $v_k \rightsquigarrow v$ include all the cycles involving $v_1, \ldots v_{k-1}$.

4.3.5 Correctness and Complexity

To prove that the transitive analysis algorithm in Figure 6 is correct, this section demonstrates that the algorithm satisfies two conditions. First, the function *Path-Summary-On-DAG* computes the correct summaries for all paths in the transformed DAG. Second, after these path summaries are extended with the pre-computed cycle information, they also include all the cycles in the original graph.

We use induction to show that each path summary $td(p, v)$ is computed correctly on the transformed DAG. Given an arbitrary vertex p in the DAG, we assume that for every vertex q closer to v in the topological order of the DAG, $td(q, v)$ is computed correctly. Because the vertices are traversed in reverse topological order of the DAG, when the algorithm computes path summaries for p, for each edge e from vertex $p \rightarrow q$, $td(q, v)$ has been already computed correctly. Since the dependence paths from p to v includes the concatenation of each edge $e : p \rightarrow q$ with $td(q, v)$, $td(p, v)$ also includes all the paths from p to v and thus is computed correctly.

To prove that each final path summary $td(p, v)$ includes all the cycles in the original graph, it is sufficient to show that all the broken cycles have been successfully recovered in $td(p, v)$. Since these cycles are broken when the twin vertices are split, they can be recovered by adding a dummy edge from each new vertex v'_i to its original vertex v_i. For each pair of twin vertices (v_i, v'_i), the pre-computed cycle information $C(v_i, v'_i)$ summarizes all the

cycles involving split vertices $v_1, .., v_i$. Thus the collection of $C(v_1, v'_1), ..., C(v_m, v'_m)$ summarizes all the cycles in the original dependence graph. After extending $td(p, v)$ with these summaries, $td(p, v)$ also includes all the original cycles. The path summary $td(p, v)$ thus is computed correctly.

Given a dependence graph G with V vertices and E edges, suppose that at most $M(M \leq V)$ vertices are split for each strongly connected component (SCC) of G. From Figure 6, the worst case complexity of finding SCCs and creating twin vertices is $O(V + E)$. The complexity of pre-computing cycle information for each SCC is $O(VM^2)$. Therefore the complexity of function *Preprocess-Cycles* is $O(V + E + VM^2)$. The worst case complexity of *Path-Summary-On-DAG* is $O(V + E)$. The complexity of recovering cycles is $O(VM)$. Therefore the worst case complexity of function *Path-Summary-On-Cyclic-Graphs* in Figure 6 is $O(V + E + VM)$.

The number of split vertices thus determines the worst case complexity of the transitive analysis algorithm in Figure 6. Although M (the largest number of split vertices in all SCCs) is $O(V)$ in the worst case (e.g., for an fully connected graph), in practice, the dependence graphs are not so densely connected. Often only a small number of vertices need to be split to break all the cycles in a SCC. Furthermore, the transitive analysis algorithm in Figure 6 can be configured with a restriction on the maximum number of split vertices. In cases where an unreasonably large number of vertices need to be split, the algorithm can simply give up and assume some *bottom* value for all the transitive path summaries. In reality, this strategy rarely degrade effectiveness of compilers because for a dependence graph that is so densely connected, usually no legal transformation is possible anyway. By bounding the value of M by a constant, both *Preprocess-Cycles* and *Path-Summaries-On-Cyclic-Graphs* would require time that is linear in the size of the graph, i.e., $O(V + E)$. Consequently, the transitive analysis algorithm would only require time $O(V^2+VE)$ to summarize transitive paths between all pairs of statements. This complexity is comparable to the complexity of standard dependence analysis algorithms. Furthermore, since transitive dependence analysis is applied only selectively when transforming non-perfectly nested loops, it is guaranteed to incur only moderate overhead when incorporated into a compiler.

5 Dependence Hoisting

This section introduces a new loop transformation, *dependence hoisting*, that facilitates the direct fusion and interchange of arbitrarily nested loops. This transformation fuses a group of arbitrarily nested loops and then shifts the fused loop to the outermost position of a code segment containing the original loops.

Given a group of loops as input for a dependence hoisting transformation, we determine the safety of fusing and shifting these loops by examining the dependence constraints on iterations of these loops. If these loops belong to a sequence of perfect loop nests in the original code, traditional loop interchange and fusion analysis for perfect loop nests [3, 35] would suffice; however, if these loops are non-perfectly nested inside one another, we perform transitive analysis on the dependence graph and use the summarized transitive dependences to determine the safety of fusing and shifting these loops. This section focuses on using transitive dependences to determine the safety of dependence hoisting transformations; traditional safety analysis for shifting and fusing perfectly nested loops is a well-understood topic and will not be discussed further.

Dependence hoisting transformation is realized by combining a sequence of traditional loop distribution, interchange and index set splitting transformations on perfectly nested loops. The complexity of applying dependence hoisting is thus equivalent to that of the corresponding sequence of sub-transformations. In the worst case, applying dependence hoisting to a loop nest takes time proportional to $N^2 + L^2D$, where N is the number of statements in the nest, L is the depth of the nest, and D is the size of the dependence graph for the nest. In average case, however, dependence hoisting requires much less time to finish. For a perfect loop nest, dependence hoisting is equivalent to a standard loop interchange on perfect loop nests followed by a single-loop distribution, in which case the required complexity is $O(N + D)$.

This section presents both the safety analysis and the actual transformation steps for dependence hoisting. Section 5.1 first introduces a new notation, computation slice, that defines the input information for a dependence hoisting transformation. Section 5.2 uses the non-pivoting LU code in Figure 1 as example to illustrate how to determine the safety of dependence hoisting. Finally, Section 5.3 presents both the analysis and transformation algorithms for dependence hoisting.

5.1 Computation Slice

The input information for a dependence hoisting transformation must specify which loops to be fused at the outermost position of a code segment and how to align these loops when performing fusion. The transformation can be seen as partitioning the original computation into slices — each slice executes a single fused loop iteration of the statements. The set of loops to be fused is thus denoted as a *computation slice* (or *slice*), and each loop to be fused is denoted as a *slicing loop*.

Each computation slice defines a dependence hoisting transformation for a code segment C. For each statement s inside C, the computation slice selects a loop ℓ surrounding s as the slicing loop for s and then selects a small integer as the *alignment factor* for the slicing loop. Statement s is also called the *slicing statement* of loop ℓ. The computation slice thus contains the following information.

- *stmt-set*: the set of statements in the slice;
- *slice-loop*(s) $\forall s \in$ *stmt-set*: for each statement s in *stmt-set*, the slicing loop for s;
- *slice-align*(s) $\forall s \in$ *stmt-set*: for each statement s in *stmt-set*, the alignment factor for *slice-loop*(s).

Given the above computation slice, a dependence hoisting transformation fuses all the slicing loops into a single loop ℓ_f at the outermost position of C s.t.

$$\forall\ \ell(s) = \textit{slice-loop}(s),\ Ivar(\ell_f(s)) = Ivar(\ell(s)) + \textit{slice-align}(s). \tag{9}$$

Here loop $\ell(s)$ is the slicing loop for s, and $Ivar(\ell(s))$ and $Ivar(\ell_f(s))$ are the index variables for loops $\ell(s)$ and $\ell_f(s)$ respectively. Equation (9) specifies that each iteration instance I of loop $\ell(s)$ is executed at iteration $I + \textit{slice-align}(s)$ of the fused loop ℓ_f after transformation.

A valid computation slice for C must satisfy the following three conditions.

- it includes all the statements in C;
- all of its slicing loops can be legally shifted to the outermost loop level;
- each pair of slicing loops $\ell_x(s_x)$ and $\ell_y(s_y)$ can be legally fused s.t. $Ivar(\ell_x) + \textit{slice-align}(s_x) = Ivar(\ell_y) + \textit{slice-align}(s_y)$.

If a computation slice satisfies all the above conditions, the corresponding dependence hoisting transformation does not violate any dependence constraint. If C is a single loop nest, the outermost loop of C can always be legally chosen as the slicing loops (with alignment 0) for all the statements in C, so C always has at least one valid computation slice.

Given a sequence of disjunct loop nests C_1,...,C_m, each nest C_i $(i = 1, ..., m)$ can have one or more valid computation slices, and the computation slices for two disjunct nests C_i and C_j must contain disjunct sets of statements. Because all the loop nests, C_1,...,C_m, are at the outermost position of the input code segment, there can be no dependence cycle connecting them. Similarly, there can be no dependence cycle connecting the disjunct computation slices. This property of disjunct computation slices can facilitate the further fusion of these slices, a transformation which will be discussed in Section 6.

5.2 Safety of Dependence Hoisting

This section uses transitive dependence information to resolve the safety of applying dependence hoisting at the outermost loop level of a given code segment C. Since each transformation is driven by a computation slice (see Section 5.1), the safety of dependence hoisting can be modeled as the validity of computation slices.

As discussed in Section 5.1, to determine whether a computation slice is valid, we must resolve the safety of two loop transformations: shifting an arbitrary loop $\ell(s)$ to the outermost loop level and fusing two arbitrary loops $\ell_x(s_x)$ and $\ell_y(s_y)$ at the outermost loop level (the fused loop will be placed at the outermost loop level). Section 5.2.1 presents the legality conditions for these two transformations. Section 5.2.2 then uses the non-pivoting LU code in Figure 1 to illustrate the safety analysis for dependence hoisting.

5.2.1 Interchange and Fusion Analysis

To decide whether an arbitrary loop $\ell(s)$ (a loop ℓ surrounding statement s) can be legally shifted to the outermost loop level, we examine the self transitive dependence $td(s, s)$ and conclude that the shifting is legal if the following equation holds:

$$\forall D \in td(s, s),\ D(\ell, \ell) = \text{“} = n\text{”}\ or\ \text{“} \leq n\text{”},\ where\ n \leq 0. \tag{10}$$

The above equation indicates that each iteration I of loop $\ell(s)$ depends only on itself or previous iterations of loop $\ell(s)$. Consequently, placing loop $\ell(s)$ at the outermost loop level of statement s is legal because no dependence cycle connecting s is reversed by this transformation.

To decide whether two loops $\ell_x(s_x)$ and $\ell_y(s_y)$ can be legally fused at the outermost loop level, we examine the transitive dependences $td(s_x, s_y)$ and $td(s_y, s_x)$. We conclude that the fusion is legal if the following equation holds:

$$\begin{aligned} &\forall D \in td(s_y, s_x), \quad Dir(D(\ell_y, \ell_x)) = \text{“}=\text{”} \ or \ \text{“}\leq\text{”} \quad and \\ &\forall D \in td(s_x, s_y), \quad Dir(D(\ell_x, \ell_y)) = \text{“}=\text{”} \ or \ \text{“}\leq\text{”} \end{aligned} \tag{11}$$

If Equation (11) holds, the two loops $\ell_x(s_x)$ and $\ell_y(s_y)$ can be fused into a single loop $\ell_f(s_x, s_y)$ s.t.

$$Ivar(\ell_f) = Ivar(\ell_x) = Ivar(\ell_y) + align, \tag{12}$$

where *align* is a small integer and is called the *alignment factor for loop* ℓ_y. The value of *align* must satisfy the following equation:

$$\begin{aligned} &a_y \leq align \leq -a_x, where \\ &\quad a_x = Max\{\ Align(D(\ell_y, \ell_x)), \quad \forall D \in td(s_y, s_x)\ \}, \\ &\quad a_y = Max\{\ Align(D(\ell_x, \ell_y)), \quad \forall D \in td(s_x, s_y)\ \}. \end{aligned} \tag{13}$$

From Equation (12), each iteration I of the fused loop $\ell_f(s_x, s_y)$ executes both the iteration $I : \ell_x(s_x)$ (iteration I of loop $\ell_x(s_x)$) and the iteration $I - align : \ell_y(s_y)$. From Equation (11) and (13), iteration $I : \ell_x(s_x)$ depends on the iterations $\leq I + a_x : \ell_y(s_y)$, which are executed by the iterations $\leq I + a_x + align$ of loop $\ell_f(s_y)$. Similarly, iteration $I - align : \ell_y(s_y)$ depends on the iterations $\leq I - align + a_y : \ell_x(s_x)$, which are executed by the same iterations of loop $\ell_f(s_x)$. Since $a_y \leq align \leq -a_x$ from Equation (13), we have $I + align + a_x \leq I$ and $I - align + a_y \leq I$. Each iteration I of the fused loop $\ell_f(s_x, s_y)$ thus depends only on itself or previous iterations. Consequently, no dependence direction is reversed by this fusion transformation.

5.2.2 Example: Non-pivoting LU

This section illustrates the safety analysis of dependence hoisting using the KJI form of non-pivoting LU in Figure 1(a). Figure 7 shows the dependence and transitive dependence information for this code using the EDM

$$\mathbf{d(s_1, s_2)} = \left\{ \begin{matrix} & k(s_2) & j(s_2) & i(s_2) \\ k(s_1) & (= 0 & \leq -1 & \leq -1) \\ i(s_1) & (\geq 1 & * & = 0) \end{matrix} \right\}$$

$$\mathbf{d(s_2, s_1)} = \left\{ \begin{matrix} & k(s_1) & i(s_1) & & k(s_1) & i(s_1) \\ k(s_2) & (\leq -1 & \leq -2) & & (\leq -1 & \leq -2) \\ j(s_2) & (= 0 & \leq -1) & & (= 0 & \leq -1) \\ i(s_2) & (\geq 1 & = 0) & & (= 0 & \leq -1) \end{matrix} \right\}$$

$$\mathbf{d(s_2, s_2)} = \left\{ \begin{matrix} & k(s_2) & j(s_2) & i(s_2) & & k(s_2) & j(s_2) & i(s_2) \\ k'(s_2) & (\leq -1 & \leq -2 & \leq -2) & & (\leq -2 & \leq -3 & \leq -3) \\ j'(s_2) & (\geq 1 & = 0 & *) & & (= 0 & \leq -1 & \leq -1) \\ i'(s_2) & (\geq 1 & * & = 0) & & (\geq 1 & * & = 0) \\ k'(s_2) & (\leq -2 & \leq -3 & \leq -3) & & & & \\ j'(s_2) & (\geq 1 & = 0 & *) & & & & \\ i'(s_2) & (= 0 & \leq -1 & \leq -1) & & & & \end{matrix} \right\}$$

(a) dependence edges

$$\mathbf{td(s_1, s_1)} = \left\{ \begin{matrix} & k(s_1) & i(s_1) & & k(s_1) & i(s_1) \\ k'(s_1) & (\leq -1 & \leq -2) & & (\leq -1 & \leq -2) \\ i'(s_1) & (\geq 1 & = 0) & & (\leq 0 & \leq -1) \end{matrix} \right\}$$

$$td(s_1, s_2) = \left\{ \begin{matrix} & k(s_2) & j(s_2) & i(s_2) & & k(s_2) & j(s_2) & i(s_2) \\ k(s_1) & (\leq 0 & \leq -1 & \leq -1) & & (\leq -1 & \leq -2 & \leq -2) \\ i(s_1) & (\geq 1 & * & = 0) & & (\leq 0 & \leq -1 & \leq -1) \end{matrix} \right\}$$

$$\mathbf{td(s_2, s_1)} = \left\{ \begin{matrix} & k(s_1) & i(s_1) & & k(s_1) & i(s_1) \\ k(s_2) & (\leq -1 & \leq -2) & & (\leq -1 & \leq -2) \\ j(s_2) & (\leq 0 & \leq -1) & & (\leq 0 & \leq -1) \\ i(s_2) & (\geq 1 & = 0) & & (\leq 0 & \leq -1) \end{matrix} \right\}$$

$$\mathbf{td(s_2, s_2)} = \left\{ \begin{matrix} & k(s_2) & j(s_2) & i(s_2) & & k(s_2) & j(s_2) & i(s_2) \\ k'(s_2) & (\leq -1 & \leq -2 & \leq -2) & & (\leq -2 & \leq -3 & \leq -3) \\ j'(s_2) & (\geq 1 & = 0 & *) & & (\leq 0 & \leq -1 & \leq -1) \\ i'(s_2) & (\geq 1 & * & = 0) & & (\geq 1 & * & = 0) \\ k'(s_2) & (\leq -2 & \leq -3 & \leq -3) & & & & \\ j'(s_2) & (\geq 1 & = 0 & *) & & & & \\ i'(s_2) & (\leq 0 & \leq -1 & \leq -1) & & & & \end{matrix} \right\}$$

(b) transitive dependence edges

Figure 7: Dependence and transitive dependence edges for non-pivoting LU

$slice_j$:
- $stmt\text{-}set = \{s_1, s_2\}$
- $slice\text{-}loop(s_1) = k(s_1)$
- $slice\text{-}align(s_1) = 0$
- $slice\text{-}loop(s_2) = j(s_2)$
- $slice\text{-}align(s_2) = 0$

$slice_k$:
- $stmt\text{-}set = \{s_1, s_2\}$
- $slice\text{-}loop(s_1) = k(s_1)$
- $slice\text{-}align(s_1) = 0$
- $slice\text{-}loop(s_2) = k(s_2)$
- $slice\text{-}align(s_2) = 0$

$slice_i$:
- $stmt\text{-}set = \{s_1, s_2\}$
- $slice\text{-}loop(s_1) = i(s_1)$
- $slice\text{-}align(s_1) = 0$
- $slice\text{-}loop(s_2) = i(s_2)$
- $slice\text{-}align(s_2) = 0$

Figure 8: Computation slices for non-pivoting LU

representation described in Section 4.1. Figure 8 shows the valid computation slices for this code. The following illustrates how to automatically construct the slices in Figure 8 using the transitive dependence information in Figure 7.

To construct valid computation slices for the non-pivoting LU code, we first select an arbitrary statement as starting point and find all the candidate slicing loops for this statement. Suppose that statement s_1 is selected. We identify candidate slicing loops for s_1 as each loop $\ell(s_1)$ that can be safely shifted to the outermost loop level. The safety is resolved by examining the self transitive dependence $td(s_1, s_1)$ in Figure 7(b). From $td(s_1, s_1)$, the self dependence conditions are "≤ -1" for loop $k(s_1)$ and "$= 0$ or ≤ -1" for loop $i(s_1)$; both conditions satisfy Equation (10) in Section 5.2.1. Both loops $k(s_1)$ and $i(s_1)$ thus can be legally shifted to the outermost loop level.

We then create two computation slices, $slice_k$ and $slice_i$, where $slice_k$ selects the slicing loop $k(s_1)$ for statement s_1, and $slice_i$ selects the slicing loop $i(s_1)$ for s_1. The alignment factors for both slicing loops are 0. Each slice is then extended with a compatible slicing loop for statement s_2. From the self transitive dependence $td(s_2, s_2)$ in Figure 7(b), all the loops surrounding s_2 (loops $k(s_2)$, $j(s_2)$ and $i(s_2)$) can be legally shifted to the outermost loop level and thus are candidate slicing loops for s_2.

To further extend the computation slice $slice_k$, we find a slicing loop $\ell(s_2)$ for statement s_2, a loop that can

be legally fused with the slicing loop $k(s_1)$ already in $slice_k$. The legality of fusion is resolved by examining the transitive dependences $td(s_1, s_2)$ and $td(s_2, s_1)$ in Figure 7(b). From $td(s_1, s_2)$, the dependence conditions for loop $k(s_1)$ are "$k(s_1) \leq k(s_2)$", "$k(s_1) \leq j(s_2) - 1$" and "$k(s_1) \leq i(s_2) - 1$"; from $td(s_2, s_1)$, the conditions are "$k(s_2) \leq k(s_1) - 1$", "$j(s_2) \leq k(s_1)$" and "$i(s_2) \geq k(s_1) + 1$ or $i(s_2) \leq k(s_1)$". The conditions between loops $k(s_1)$ and $k(s_2)$ satisfy Equation (11) in Section 5.2.1; thus these two loops can be legally fused. From Equation (13), the alignment factor for loop $k(s_2)$ is 0 or 1. Similarly, loop $j(s_2)$ can be fused with loop $k(s_1)$ with fusion alignment 0 or -1. Loop $i(s_2)$ cannot be fused with loop $k(s_1)$ because of the dependence condition "$i(s_2) \geq k(s_1) + 1$" in $td(s_2, s_1)$.

We then duplicate $slice_k$ with another slice $slice_j$, extend $slice_k$ by selecting the slicing loop $k(s_2)$ for s_2, and then extend $slice_j$ by selecting the slicing loop $j(s_2)$ for s_2. The completed slices are shown in Figure 8. Figure 8 also shows the completed slice $slice_i$ which initially selects the slicing loop $i(s_1)$ for statement s_1. Here loop $i(s_2)$ can be legally fused with loop $i(s_1)$ and thus is selected as the slicing loop for statement s_2.

The three slices in Figure 8 can be used to freely translate between any two of the three loop orderings of non-pivoting LU in Figure 1. Section 2 has illustrated how to translate Figure 1(a) to (c) using $slice_j$. Similarly, using $slice_i$ can translate (a) to (b), and using $slice_k$ can translate (c) to (a). Section 5.3.2 describes the detailed algorithm to establish the translations.

5.3 Dependence Hoisting Algorithms

This section presents the algorithms both for systematically constructing valid computation slices and for using computation slices to drive dependence hoisting transformations. Figure 9 summarizes the algorithms for both dependence hoisting analysis and transformation. Sections 5.3.1 and 5.3.2 elaborate the functions in Figure 9. Section 5.3.3 then presents the correctness proof and complexity of the algorithm.

5.3.1 Dependence Hoisting Analysis

The safety analysis of dependence hoisting is encoded by the function *Hoisting-Analysis* in Figure 9. This function constructs all the valid computation slices for an input loop nest C and then puts these slices into the output variable set *slice-set*. The algorithm is separated into three steps.

Hoisting-Analysis(C, *slice-set*)
(1) *slices-in-construction* $= \emptyset$
s_0 = an arbitrary statement in C
For each loop $\ell_0(s_0)$ that can be shifted outermost
Add a new slice *slice* into *slices-in-construction*
Add $\ell_0(s_0)$ into *slice*:
slice-loop$(s_0) = \ell_0(s_0)$, *slice-align*$(s_0) = 0$
(2) If *slices-in-construction* $= \emptyset$, stop.
slice = Remove a slice from *slices-in-construction*
(3) For each statement s in C but not in *slice*
For each loop $\ell(s)$ that can be shifted outermost
and can be fused with other loops in *slice*
align = alignment factor for $\ell(s)$
If (*slice-loop*$(s) = \emptyset$ in *slice*)
Add $\ell(s)$ into *slice*:
slice-loop$(s) = \ell(s)$
slice-align$(s) =$ *align*
Else $slice_1 = Duplicate$(slice)
Add $\ell(s)$ into $slice_1$:
slice-loop$(s) = \ell(s)$
slice-align$(s) =$ *align*
Add $slice_1$ into *slices-in-construction*
If (*slice* covers C), add *slice* to *slice-set*
Goto step(2)

Hoisting-Transformation(C, *slice*)
(1) $range = \bigcup_{s \in C}(Range(\text{slice-loop}(s)) + \text{slice-align}(s))$
ℓ_f = create a new loop with $Range(\ell_f) = range$
Put ℓ_f surrounding C
For each statement $s \in slice$ with $\ell = \text{slice-loop}(s)$
$cond(s)$= "if $(Ivar(\ell_f) = Ivar(\ell) + \text{slice-align}(s))$"
Insert $cond(s)$ surrounding s
Modify the dependence graph of C.
(2) For each statement $s \in slice$ with $\ell(s) = \text{slice-loop}(s)$
Distribute $\ell(s)$ to enclose only s:
(2.1) For each s_1 strongly connected with s in $Dep(\ell)$
$m = \text{slice-align}(s_1) - \text{slice-align}(s)$
$\ell_1(s_1) = \text{slice-loop}(s_1)$
Split s_1 : $Ivar(\ell) < Ivar(\ell_1) + m$,
$Ivar(\ell) = Ivar(\ell_1) + m$,
$Ivar(\ell) > Ivar(\ell_1) + m$.
Modify the dependence graph of C
(2.2) Distribute loop $\ell(s)$
While ($\ell(s)$ contains other stmts than s) do
Interchange loop $\ell(s)$ with the loop inside it
Modify dependence graph of C
Distribute loop $\ell(s)$
(3) For each statement $s \in slice$ with $\ell(s) = \text{slice-loop}(s)$
Replace $Ivar(\ell)$ inside loop $\ell(s)$
Adjust loop ranges between ℓ_f and $\ell(s)$
Remove $cond(s)$ and $\ell(s)$

Figure 9: Dependence hoisting algorithms

Step (1) of the algorithm starts from an arbitrary statement s_0 in the input nest C and first finds all the candidate slicing loops for s_0: each slicing loop $\ell(s_0)$ must be legal to be shifted to the outermost loop level. For each found candidate slicing loop $\ell_0(s_0)$, this step creates a computation slice including $\ell_0(s_0)$ and then puts the created partial slice into a variable set *slices-in-construction* for further examination.

Step(2) of the algorithm examines the variable set *slices-in-construction* and terminates the algorithm if this set is empty; otherwise, it removes a slice from *slices-in-construction* and tries to complete the slice by going to step(3).

Step (3) first completes the partial slice *slice* selected by step (2) and then goes back to step (2) for more partial slices. For each statement s not yet in the partial slice *slice*, this step identifies valid slicing loops for s as each loop $\ell(s)$ that can be shifted to the outermost loop level and can be fused with the slicing loops already in *slice* (how to determine these conditions is described in section 5.2.1). For each found slicing loop $\ell(s)$, an alignment factor *align* is computed from the fusion alignments between loop $\ell(s)$ and other slicing loops in *slice*. If multiple slicing loops are found, step (3) duplicates the original slice to remember the extra slicing loops and then adds the duplicated slices into the variable set *slices-in-construction* for further examination. At the end of step (3), if all the statements have been included in *slice*, this slice is completed successfully and thus can be collected into the output variable *slice-set*; otherwise, this slice is thrown away.

5.3.2 Dependence Hoisting Transformation

The dependence hoisting transformation algorithm is encoded by the function *Hoisting-Transformation* in Figure 9. Given a computation slice *slice*, this function transforms the input code segment C in the following three steps.

Step (1) of the transformation puts a new outermost dummy loop ℓ_f surrounding C. The new loop ℓ_f unions the iteration ranges of all the slicing loops after proper alignments. This step then inserts a conditional $cond(s)$ surrounding each statement s in *slice*. The inserted conditional $cond(s)$ forces statement s to execute only if $Ivar(\ell_f) = Ivar(\textit{slice-loop}(s)) + \textit{slice-align}(s)$. Finally, this step modifies the dependence graph of C to include dependence conditions for the new loop ℓ_f: for each statement s, $Ivar(\ell_f(s))$ has the same dependence conditions as those of $Ivar(\textit{slice-loop}(s)) + \textit{slice-align}(s)$. Figure 2(b) in Section 2 illustrates this modification using the KJI form of non-pivoting LU.

For each slicing statement s, step (2) of the transformation then distributes the slicing loop $\ell(s)$ so that $\ell(s)$ encloses only its slicing statement s. To achieve this, step (2.1) first applies loop index-set splitting to remove all the dependence cycles that are incident to statement s and are carried by loop ℓ. For each statement s_1 that is strongly connected with s in $Dep(\ell)$ (the dependence graph of ℓ), step (2.1) splits s_1 into three statements under the execution conditions $Ivar(\ell) < Ivar(\ell_1) + m$, $Ivar(\ell) = Ivar(\ell_1) + m$ and $Ivar(\ell) > Ivar(\ell_1) + m$ respectively, where $\ell_1(s_1)$ is the slicing loop for s_1, and $m = \textit{slice-align}(s_1) - \textit{slice-align}(s)$. Step (2.2) then repeatedly applies loop distribution and interchange to remove all the dependence cycles that are incident to s and are carried by loops inside $\ell(s)$. This step first distributes loop $\ell(s)$ so that $Dep(\ell)$ is strongly connected. If $\ell(s)$ still encloses statements other than s, this step interchanges $\ell(s)$ with the loop immediately inside $\ell(s)$ and then tries again. Since steps (2.1) and (2.2) successfully remove all the dependence cycles incident to s in $Dep(\ell)$, eventually the distributed loop $\ell(s)$ encloses only statement s. For proof of these two steps, see section 5.3.3.

Step (3) of the transformation then removes all the distributed slicing loops and the conditionals that synchronize these loops with the new outermost loop ℓ_f. Before removing each slicing loop $\ell(s)$, this step makes two adjustments to the original loop nest: first, it replaces all the appearances of loop index variable $Ivar(\ell(s))$ inside loop $\ell(s)$ with expression $Ivar(\ell_f) - \textit{slice-align}(s)$; second, it adjusts the iteration ranges of the loops between ℓ_f and $\ell(s)$ to ensure that the correct iteration set of statement s is executed. Step (3) then removes

both the slicing loop $\ell(s)$ and the conditional *cond*(s) because they have become redundant.

5.3.3 Correctness and Complexity

The hoisting analysis algorithm in Figure 9 is correct in that only valid computation slices are collected. Each collected slice satisfies three conditions: first, it includes all the statements in the input loop nest; second, all the slicing loops can be legally shifted to the outermost loop level; third, each pair of slicing loops $\ell_x(s_x)$ and $\ell_y(s_y)$ can be legally fused after being aligned with *slice-align*(s_x) and *slice-align*(s_y) respectively.

The correctness of the hoisting transformation algorithm in Figure 9 is proved by demonstrating that the transformed code at each step satisfies two conditions: first, each statement still executes the same set of iteration instances; second, no dependence is reversed in the dependence graph.

Step (1) of the dependence hoisting transformation puts a new loop ℓ_f at the outermost position and then inserts conditionals surrounding the slicing statements. Each conditional *cond*(s) synchronizes the slicing loop *slice-loop*(s) with the outermost loop ℓ_f in a lock-step fashion, thus forcing statement s to execute the original set of iteration instances. In addition, the inserted conditionals shift the loop-carrying levels of a set of dependences to the outermost loop ℓ_f. Since all the slicing loops can be legally shifted to the outermost loop level and can be legally fused with each other, no dependence direction is reversed.

Step (2) of the transformation algorithm distributes each slicing loop $\ell(s)$ so that $\ell(s)$ encloses only its slicing statement s. After step (1), all the dependence paths from statement s_x to s_y are now carried by the new outermost loop ℓ_f unless the following condition holds:

$$Ivar(\ell_x) + \textit{slice-align}(s_x) = Ivar(\ell_y) + \textit{slice-align}(s_y) \tag{14}$$

Here $\ell_x = \textit{slice-loop}(s_x)$ and $\ell_y = \textit{slice-loop}(s_y)$. Thus only the dependence paths satisfying Equation (14) need to be considered when distributing the slicing loops.

For each slicing loop $\ell(s)$, step (2.1) applies loop index-set splitting to remove all the dependence cycles that are incident to statement s and are carried by loop ℓ in $Dep(\ell)$ (the dependence graph of ℓ). For each statement $s_1 \neq s$ inside loop ℓ, suppose $\textit{slice-loop}(s_1) = \ell_1$. From Equation (14), each dependence EDM D between statements s_1 and s satisfies the condition $D(\ell(s), \ell_1(s_1)) =$ "$= m$", where $m = \textit{slice-align}(s_1) - \textit{slice-align}(s)$. Once statement s_1 is split into three statements s_1', s_1'' and s_1''' under execution conditions $Ivar(\ell) < Ivar(\ell_1)+m$,

$Ivar(\ell) = Ivar(\ell_1) + m$ and $Ivar(\ell) > Ivar(\ell_1) + m$ respectively, each dependence EDM between statements s_1' and s satisfies the condition $Ivar(\ell(s_1')) \leq Ivar(\ell(s)) - 1$. Since each iteration of loop ℓ can depend only on itself or previous iterations in $Dep(\ell)$, there is no dependence path from statement s to s_1' in $Dep(\ell)$. Similarly, there is no dependence path from statement s_1''' to s in $Dep(\ell)$. Thus all the dependence cycles between statements s and s_1 are now between s and s_1''. Since each dependence EDM between s_1'' and s satisfies the condition $Ivar(\ell(s)) = Ivar(\ell(s_1''))$, no dependence cycle connecting s is carried by loop ℓ.

For each slicing loop $\ell(s)$, step (2.2) of the dependence hoisting transformation further removes from $Dep(\ell)$ all the dependence cycles that are incident to statement s and are carried by loops inside $\ell(s)$. This step first distributes loop $\ell(s)$ so that $\ell(s)$ does not carry any dependence. If $\ell(s)$ still encloses statements other than s, there must be dependence cycles carried by a loop ℓ' perfectly nested inside $\ell(s)$. step (2.2) can then shift loop ℓ inside ℓ' to remove these dependence cycles from $Dep(\ell)$. The distribution and interchange repeat until loop $\ell(s)$ encloses only statement s.

Since each slicing loop $\ell(s)$ now encloses only statement s and the conditional synchronizing $\ell(s)$ with the outermost loop ℓ_f, both the slicing loop $\ell(s)$ and the conditional $cond(s)$ can be safely removed. Step(3) of the dependence hoisting transformation removes both the slicing loops and conditionals while making sure each statement still executes the same set of iteration instances. Therefore all the transformation steps are legal.

We now determine the complexity of the dependence hoisting algorithms. Given an input loop nest C, suppose that there are N statements in C, the loop nesting depth of C is L, and the size of the dependence graph of C is D. From the function *Hoisting-Analysis* in Figure 9, at most L computation slices can be constructed, and the construction of each computation slice examines all pairs of the statements in C. Therefore the worst case complexity for this function is $O(N^2L)$. From the function *Hoisting-Transformation* in Figure 9, step (2.1) makes at most N^2 attempts to split the iteration ranges of slicing loops, and step (2.2) makes at most L attempts to distribute each slicing loop. For step (2.2), the worst case happens when all the slicing loops are nested inside one another and when each slicing loop $\ell(s)$ needs to be shifted to the innermost position before it can be successfully distributed. In this case, there are at most L different slicing loops in the input computation slice, so the worst case complexity of the *Hoisting-Transformation* function is $O(N^2 + L^2D)$.

Although the worst case complexity of a single dependence hoisting transformation is $O(N^2 + L^2D)$, in

average case, the complexity of the transformation is much lower. In reality, most slicing loops can be successfully distributed at the first effort, and the distribution of one slicing loop often automatically results in the distribution of other slicing loops as well. For example, in the case where the group of slicing loops is a single loop that is perfectly nested within the original outermost loop, the transformation needs only to create a new outermost loop ℓ_f, distribute all the slicing loops at once with one loop distribution, and then remove the distributed loops and conditionals. The complexity thus becomes $O(N + D)$.

6 Transformation Framework

This section develops a framework that systematically combines the dependence hoisting transformation introduced in Section 5 with other simple techniques to efficiently optimize applications for better locality. The framework hierarchically considers code segments at different loop levels and applies the same transformation algorithms at the outermost loop level of each code segment, guaranteeing that optimizations are applied at all loop levels and thus no generality is sacrificed.

To achieve better locality, our framework uses dependence hoisting to facilitate three optimizations on arbitrary loops: loop fusion, interchange and blocking. The framework first constructs all the valid computation slices for an input code segment. It then arranges the best nesting order for the constructed computation slices and fuses disjunct computation slices when profitable. Finally, it applies the rearranged computation slices to perform dependence hoisting transformations on the original code, while combining dependence hoisting with loop strip-mining to achieve loop blocking when profitable. Simple data-reuse analysis is performed to resolve the profitability of the fusion, interchange and blocking optimizations.

Note that our framework is only a prototype implementation to demonstrate the benefit of integrating dependence hoisting with traditional transformation techniques. For example, although dependence hoisting is necessary only when transforming non-perfect loop nests, this framework substitutes dependence hoisting for both loop interchange and fusion transformations in all cases. A production compiler can easily adopt a more selective strategy and apply dependence hoisting only when required. The production compiler can also incorporate more sophisticated profitability analysis models such as those by Kennedy and McKinley [19]. These models

can be integrated into our framework in a straightforward fashion and will not be further discussed.

To present the framework in more detail, Section 6.1 first defines some notations to classify different groups of computation slices. Section 6.2 describes how to use dependence hoisting to achieve aggressive loop interchange, fusion, and blocking optimizations. Section 6.3 then presents the framework that systematically applies dependence hoisting to optimize applications for locality.

6.1 Classifying Computation Slices

This section defines notations to classify different groups of computation slices. As discussed in Section 5, each dependence hoisting transformation is driven by a computation slice, which contains information necessary to fuse a set of loops and then shift the fused loop to the outermost position of some code segment. Each computation slice thus can be seen as representing a single loop, denoted as the *fused loop* of the computation slice. Because modifications to each computation slice immediately result in the corresponding changes to its fused loop after a dependence hoisting transformation, traditional notations for loops can be extended to computation slices as well.

Two computation slices, $slice_1$ and $slice_2$, are defined to be nested if both slices contain the same set of statements; that is, the fused loops of these two slices will be nested inside one another after dependence hoisting transformations. Given a set of slices $slice\text{-}set = \{slice_1, ..., slice_m\}$, if all the slices contain the same set of statements, this group of slices is denoted as a *slice nest*; that is, the fused loops of these slices will form a loop nest after dependence hoisting transformations. The nesting order of these slices is defined as the nesting order of the corresponding fused loops.

Two computation slices, $slice_1$ and $slice_2$, are defined to be disjunct if they contain disjunct sets of statements. Similarly, a sequence of computation slices $slice_1$, ..., $slice_m$ is defined to be disjunct if every pair of computation slices in the sequence is disjunct. Given a sequence of disjunct computation slices, there can be no dependence cycle connecting these slices (see Section 5.1). These slices thus can be further fused to achieve the fusion of different loop nests in the original code. A fusion algorithm is presented in Section 6.2.2.

Comp-Slice-Fusible(Dep, $slice_1$, $slice_2$, $align$)
  $align = -\infty$
  For (each dependence EDM D
       from $s_x \in slice_1$ to $s_y \in slice_2$)
    $\ell_x = slice_1$:*slice-loop*(s_x);
    $\ell_y = slice_2$:*slice-loop*(s_y)
    If ($Dir(D(\ell_x, \ell_y)) \neq$ " = " or $\leq$), return false
    $align_y = Align(D(\ell_x, \ell_y))$ +*slice-align*(s_x)
                                  −*slice-align*(s_y)
    $align = max(align, align_y)$
  return true

Fuse-Comp-Slice($slice_1$, $slice_2$, $align$)
  $slice$ = create a new computation slice
  For (each statement $s \in slice_1$) do
    Add s into $slice$:
        slice-loop$(s) = slice_1$: *slice-loop*(s);
        slice-align$(s) = slice_1$: *slice-align*(s)
  for (each statement $s \in slice_2$) do
    Add s into $slice$:
        slice-loop$(s) = slice_2$: *slice-loop*(s);
        slice-align$(s) = slice_2$: *slice-align*$(s) + align$
  return $slice$

Figure 10: Fusing computation slices

6.2 Achieving Interchange, Fusion, And Blocking

This section illustrates how to use computation slices to achieve loop interchange, fusion, and blocking optimizations. The following subsections describe how to achieve each of the transformations respectively.

6.2.1 Achieving Loop Interchange

This section describes how to realize loop interchange by rearranging a nest of computation slices. Here because each slice represents a collection of loops that can be fused into a single loop, by interchanging the order of applying two slices, we directly interchange the nesting order of the original two groups of slicing loops.

Given two nested computation slices $slice_1$ and $slice_2$, to shift the slicing loops in $slice_2$ outside of the ones in $slice_1$, we first use $slice_1$ to perform a dependence hoisting transformation. We then use the fused loop ℓ_{f1} of $slice_1$ as the input code for another dependence hoisting transformation using $slice_2$, which in turn shifts the fused loop of $slice_2$ outside loop ℓ_{f1}.

In general, to achieve the desired nesting order of m computation slices, $slice_1$, $slice_2$, ..., $slice_m$, we first use $slice_m$ to drive a dependence hoisting transformation, which shifts the fused loop ℓ_m of $slice_m$ to the outermost loop level. We then use $slice_{m-1}$ to shift the fused loop ℓ_{m-1} of $slice_{m-1}$ outside of ℓ_m, and so on. The desired nesting order of m computation slices thus can be achieved using m dependence hoisting transformations.

6.2.2 Achieving Loop Fusion

This section describes how to fuse two disjunct computation slices. Because each slice contains a group of loops that can be shifted to the outermost loop level, fusing two disjunct slices automatically achieves the fusion of the slicing loops in both slices.

Figure 10 presents two algorithms: one determines whether two computation slices can be legally fused, and

the other performs the actual fusion of the two computation slices.

The function *Comp-Slice-Fusible* in Figure 10 determines whether two disjunct computation slices can be legally fused. Because there is no dependence cycle connecting the two slices (see Section 6.1), this function assumes that there are only dependence edges from $slice_1$ to $slice_2$ in the dependence graph Dep (if the opposite is true, the two slice arguments can be switched without sacrificing any generality). The algorithm examines each dependence edge D from statement $s_x \in slice_1$ to statement $s_y \in slice_2$. If the dependence condition from loop ℓ_x ($slice\text{-}loop(s_x)$ in $slice_1$) to loop ℓ_y ($slice\text{-}loop(s_y)$ in $slice_2$) has a direction that is neither $=$ nor $\leq$, the dependence edge will be reversed after fusion, and the fusion is not legal; otherwise, the dependence edge does not prevent the two slicing loops from being fused, in which case the algorithm restricts the fusion alignment factor $align$ for $slice_2$ so that

$$align \geq Align(D(\ell_x, \ell_y)) + slice\text{-}align(s_x) - slice\text{-}align(s_y). \tag{15}$$

If the algorithm succeeds in finding a valid fusion alignment $align$ after examining all the dependence edges, the two computation slices should be fused so that

$$Ivar(\ell_{f1}) = Ivar(\ell_{f2}) + align. \tag{16}$$

where ℓ_{f1} and ℓ_{f2} represent the fused loops of $slice_1$ and $slice_2$ respectively. Thus $slice_2$ needs to be aligned by the factor $align$ before being fused with $slice_1$.

To prove that the fusion algorithm in Figure 10 is correct, the following shows that after fusing $slice_1$ and $slice_2$ according to Equation (16), no dependence edge from $slice_1$ to $slice_2$ is reversed in the dependence graph. First, for each pair of slicing loops, $\ell_x(s_x) \in slice_1$ and $\ell_y(s_y) \in slice_2$, the following equations hold from the definition of valid computation slices (see Section 5.1):

$$Ivar(\ell_{f1}) = Ivar(\ell_x) + slice\text{-}align(s_x) \tag{17}$$

$$Ivar(\ell_{f2}) = Ivar(\ell_y) + slice\text{-}align(s_y) \tag{18}$$

After substituting the above two equations for ℓ_{f1} and ℓ_{f2} in Equation (16), the following relation between the slicing loops ℓ_x and ℓ_y is satisfied:

$$Ivar(\ell_x) + slice\text{-}align(s_x) = Ivar(\ell_y) + slice\text{-}align(s_y) + align \tag{19}$$

Now consider each dependence EDM D from s_x to s_y. Because the fusion alignment *align* satisfies Equation (15), substituting this inequality for *align* in Equation (19) obtains

$$\begin{aligned} Ivar(\ell_x) + \textit{slice-align}(s_x) \quad \leq \quad & Ivar(\ell_y) + \textit{slice-align}(s_y) + \\ & Align(D(\ell_x, \ell_y)) + \textit{slice-align}(s_x) - \textit{slice-align}(s_y), \end{aligned}$$

which is equivalent to

$$Ivar(\ell_x) \leq Ivar(\ell_y) + Align(D(\ell_x, \ell_y)) \tag{20}$$

The above equation indicates that the original dependence condition between ℓ_x and ℓ_y is maintained after fusing $slice_1$ and $slice_2$. Therefore no dependence direction will be reversed by the fusion transformation.

The function *Fuse-Comp-Slice* in Figure 10 performs the actual fusion of the two computation slices $slice_1$ and $slice_2$. The algorithm first creates a new empty computation slice and then clones both $slice_1$ and $slice_2$ into the new slice. Before adding each statement s of $slice_2$ into the new slice, the algorithm adjusts the slicing alignment factor for s with the alignment factor *align* so that the fusion relation specified by Equation (15) is satisfied.

6.2.3 Achieving Loop Blocking

Traditionally, loop blocking is achieved by combining loop interchange with loop strip-mining. Given a nest of computation slices, substituting dependence hoisting for loop interchange, we thus automatically achieve loop blocking by combining loop strip-mining with dependence hoisting.

To block an input loop nest C using a nest of computation slices, *slice-nest*, we remove each slice from *slice-nest* in the reverse of the desired nesting order for these slices. After using each slice to drive a dependence hoisting transformation, we strip-mine the new fused loop ℓ_f into a strip-counting loop ℓ_c and a strip-enumerating loop ℓ_t. We then use loop ℓ_t as the input loop nest for further dependence hoisting transformations, which in turn will shift a new set of loops outside loop ℓ_t but inside loop ℓ_c, thus blocking loop ℓ_f.

As an example, Figure 11 illustrates the steps to block the KJI form of non-pivoting LU in Figure 1(a). Figure 8 shows the three valid computation slices for this code: $slice_i = \{i(s_1), i(s_2)\}$, $slice_j = \{k(s_1), j(s_2)\}$, and $slice_k = \{k(s_1), k(s_2)\}$ (the alignment factors for all the slicing loops are 0).

```
do x_i = 2, n, b
do i = x_i, min(x_i + b - 1, n)
  do k = 1, i - 1
s1: a(i,k) = a(i,k)/a(k,k)
    do j = k + 1, n
s2: a(i,j) = a(i,j) - a(i,k) * a(k,j)
    enddo
  enddo
enddo
enddo
```

(a) IKJ

```
do x_j = 1, n, b
do x_i = 2, n, b
do j = x_j, min(x_j + b - 1, n)
  do i = x_i, min(x_i + b - 1, n)
    do k = 1, min(i - 1, j - 1)
s2: a(i,j) = a(i,j) - a(i,k) * a(k,j)
    enddo
  enddo
  do i = max(x_i, j + 1),
         min(x_i + b - 1, n)
s1: a(i,j) = a(i,j)/a(j,j)
  enddo
enddo
enddo
enddo
```

(b) JIK

```
do x_j = 1, n, b
do x_i = 2, n, b
do k = 1, min(x_j + b - 1, x_i + b - 1)
  if (k ≥ x_j) then
    do i = max(x_i, k + 1),
           min(x_i + b - 1, n)
s1: a(i,k) = a(i,k)/a(k,k)
    enddo
  endif
  do j = max(x_j, k + 1),
         min(x_j + b - 1, n)
    do i = max(k + 1, x_i),
           min(x_i + b - 1, n)
s2: a(i,j) = a(i,j) - a(i,k) * a(k,j)
    enddo
  enddo
enddo
enddo
enddo
```

(c) KJI

Figure 11: Blocking non-pivoting LU

To block the non-pivoting LU code in Figure 1(a), we first use $slice_i$ to drive a dependence hoisting transformation and obtain the IKJ loop ordering of non-pivoting LU in Figure 1(b). We then strip-mine the fused loop of $slice_i$, as shown in Figure 11(a). Next, we use $slice_j$ to shift the $j(s_2)$ and $k(s_1)$ loops outside the strip-enumerating i loop in Figure 11(a), and then again strip-mine the new outermost j loop. The result of this second step is a blocked JIK loop ordering shown in Figure 11(b). Finally, we use $slice_k$ to further shift loops $j(s_1)$ and $k(s_2)$ outside the strip-enumerating j loop, and the result is show in Figure 11(c). The code in Figure 11(c) has the same loop ordering as that of the original KJI form in Figure 1(a) except that the loops $j(s_2)$, $k(s_1)$, $i(s_1)$ and $i(s_2)$ are now blocked. In Figure 11(c), both the row and column directions of the matrix are blocked, as shown in the pictorial illustration.

Note that the conditional "$if\ (k \geq x_j)$" in Figure 11(c) can be removed by splitting the iteration range of the k loop. The splitting can be achieved by integrating a loop index-set splitting step into the cleanup phase of the transformation framework, as discussed in Section 6.3.1.

6.3 Transformation Framework

This section presents our transformation framework, which systematically applies dependence hoisting to transform arbitrarily nested loops for better locality. As shown in Figure 12, the algorithm recursively invokes itself

Optimize-Code-Segment(C)
(1) *slice-nest-vec* $= \emptyset$; Distribute loop nests in C
  for (each distributed loop nest L) do
    Hoisting-Analysis(L, *slice-nest*)
    Arrange the best nesting order for *slice-nest*
    Add *slice-nest* to *slice-nest-vec*
(2) Apply typed-fusion to *slice-nest-vec*
  For (each clustered group *slicenest-group*) do
    For (each fusible pair of vertices $slicenest_1$ and $slicenest_2$)
      For (each $slice_1 \in slicenest_1$ and $slice_2 \in slicenest_2$)
        if (**Comp-Slice-Fusible**($slice_1$, $slice_2$,*align*) and
            fusion-profitable($slice_1$,$slice_2$))
          Fuse-Comp-Slice($slice_1$,$slice_2$,*align*)
(3) For (each *slice-nest* $\in$ *slice-nest-vec*) do
  C_1 = Distribute(C, *slice-nest*)
  For (each *slice* $\in$ *slice-nest* in reverse nesting order)
    Hoisting-Transformation(C_1, *slice*)
    C_1 = the new fused loop ℓ_f of *slice*
    If (blocking *slice* is profitable) then
      strip-mine ℓ_f into ℓ_c and ℓ_t
      Shift ℓ_c to the outermost loop level of C
      $C_1 = \ell_t$
  C_2 = code segment inside *slice-nest*
  if (C_2 needs to be further optimized)
    Optimize-Code-Segment(C_2)

Figure 12: Transformation steps of computation slicing

to hierarchically optimize the original code at different loop levels. Section 6.3.1 first elaborates each step of the algorithm. Section 6.3.2 then discusses the correctness and complexity of the framework.

6.3.1 Transformation Steps

Given a code segment C to optimize, the algorithm in Figure 12 optimizes C in the following three steps.

Step(1) At this step, the algorithm identifies all the valid computation slices for C. These slices are constructed by invoking the function *Hoisting-Analysis* defined in Figure 9 (note that if the input code C is a perfect loop nest, traditional loop interchange analysis would suffice). To reduce the overhead of transitive dependence analysis, the framework first performs maximum loop distribution on the input code segment C before applying dependence hoisting analysis to each of the distributed loop nest. These distributed loop nests will be re-fused at Step(3) after the collected computation slices are merged.

After identifying valid computation slices for each loop nest, the framework resolves the best nesting order for the constructed slice nest by counting the reuses of data accesses among statements. By counting the number of data reuses carried by the slicing loops of each slice, the framework arranges the nesting order so that the slices that carry more reuses will be nested inside [33, 2].

Step(2) After collecting all the valid computation slices into a vector of slice nests and arranging the best nesting order for each nest, the framework constructs a slice-fusion dependence graph, where each vertex of the graph is a nest of computation slices, and an edge e is put from vertex v_1 to v_2 if there are dependences from statements in v_1 to statements in v_2. An edge $e : v_1 \rightsquigarrow v_2$ in the fusion graph is annotated as a *bad edge* if

the two slice nests, v_1 and v_2, cannot be legally fused; that is, after applying the function *Comp-Slice-Fusible* (defined in Figure 10) to each pair of computation slices, $slice_1 \in v_1$ and $slice_2 \in v_2$, no pair of slices can be legally fused.

The framework then applies the traditional typed-fusion algorithm [24, 20] to the constructed fusion graph of *slice-nest-vec*. The typed-fusion algorithm (a linear algorithm) then in turn aggressively clusters the vertices that are not connected by fusion-preventing *bad* paths in the fusion graph. For each set of vertices clustered by the typed-fusion algorithm, the framework then proceeds to merge as many slice nests in the cluster as possible. For each pair of slice nests, $slicenest_1$ and $slicenest_2$, that can be collapsed into a vertex without creating cycles in the cluster, the framework examines every pair of slices, $slice_1 \in slicenest_1$ and $slice_2 \in slicenest_2$. If the fusion of $slice_1$ and $slice_2$ is both legal and profitable, the two slices are fused by applying the function *Fuse-Comp-Slice* defined in Figure 10. If at least one pair of slices is fused successfully, the framework collapses $slicenest_1$ and $slicenest_2$ into a single vertex in the fusion dependence graph.

When determining the profitability of fusing the two computation slices, $slice_1$ and $slice_2$, we consider both the data reuses carried by the two slices and the desired loop levels of the two slices. The desired nesting orders for both $slicenest_1$ and $slicenest_2$ have been resolved by Step(1) of the algorithm. If $slice_1$ and $slice_2$ were arranged to be at the same loop level and if more data reuses can be gained from fusing them, the fusion is profitable; otherwise, the data reuses gained by fusion is compared against the locality lost from changing the original nesting order of slices arranged by Step(1), and the fusion is permitted only if it improves the overall performance.

Note that when two slice nests are partially fused, that is, when some slices in $slicenest_1$ and $slicenest_2$ did not participate in the fusion, these left-over slices are thrown away from the collapsed vertex, and the loops corresponding to these left-over slices will be nested inside the other slices that remain in the collapsed vertex. Since the left-over slices can no longer be shifted outside the strip-mined loops of the fused slices, partial fusion can prevent the blocking of the original slice nests. To guarantee the profitability of partial fusion, we partially fuse two slice nests only when blocking is not favorable.

Step(3) The final step of the framework uses each slice nest in *slice-nest-vec* to transform the original code segment C. For each slice nest *slice-nest*, the algorithm first invokes the function *Distribute(C, slice-nest)*, which distributes C and returns a sequence of loop nests that contain exactly the statements in *slice-nest*. The returned code segment C_1 is then used as input to the dependence hoisting transformations driven by *slice-nest*.

The algorithm uses the variable C_1 to keep track of the input code segment for each dependence hoisting transformation. After applying each computation slice $slice_i$ to perform a dependence hoisting transformation, if the new fused loop ℓ_f of $slice_i$ should not be blocked, the algorithm sets C_1 to be ℓ_f so that further hoisting transformations will shift loops outside of ℓ_f; otherwise, the algorithm strip-mines ℓ_f into a strip-counting loop ℓ_c and a strip-enumerating loop ℓ_t. It then uses loop ℓ_t as the input code for further dependence hoisting transformations, which in turn will shift a new set of loops outside loop ℓ_t but inside loop ℓ_c, thus blocking loop ℓ_f. After strip-mining ℓ_f at the outermost level of C_1, the algorithm also shifts the strip-counting loop ℓ_c to the current outermost level of the original code segment C. This step guarantees that ℓ_c is nested outside the strip-counting loops obtained earlier, thus achieving the desired nesting order for the strip-counting loops as well.

After transforming the original code C through a sequence of dependence hoisting transformations, the algorithm then performs a cleanup step, which applies loop index-set splitting to eliminate the conditionals that still remain in the transformed code, as illustrated in the blocked non-pivoting LU code in Figure 11(c). The framework then recursively invokes itself to further optimize the code segment inside each optimized loop nest. By hierarchically applying the same optimizations at multiple loop levels of the original code, the algorithm guarantees the generality of the transformation framework.

6.3.2 Correctness and Complexity

The correctness of the dependence hoisting framework follows directly that of the dependence hoisting analysis and transformation algorithms, as discussed in Section 5.3. Since the blocking and fusion algorithms using dependence hoisting follow the traditional blocking and fusion theory, they also guarantee that no dependence edge is reversed. Thus no further correctness proof is necessary.

The transformation algorithms of the framework are not optimal in that the optimized code is not guaranteed to have the best performance. The framework can be further extended to incorporate more sophisticated

profitability analyses such as those described by Kennedy and McKinley [19]. It can also integrate traditional loop transformation techniques, such as reversal, index-set splitting and skewing [25, 10], in a more sophisticated fashion to achieve better performance.

The complexity of the framework is that of framework's top-level function *Optimize-Code-Segment*, defined in Figure 12. This function optimizes each statement at most L times, where L is the maximum depth of loop nests in the original code. At each loop level, the worst case complexity of applying *Hoisting-Analysis* is $O(N^2L)$ (see Section 5.3.3), where N is the number of statements in a loop nest. The worst case complexity of the first step of *Optimize-Code-Segment* is thus $O(N^2L^2)$. Since the typed-fusion algorithm has a complexity linear to the size of the fusion graph, the second step of the algorithm has the complexity $O(DL^2)$, where D is the size of the dependence graph. Since the worst case complexity of the dependence hoisting transformation is $O(N^2 + L^2D)$ (see Section 5.3.3), the third step of the algorithm has the worst case complexity $O(N^2L + L^3D)$. The worst case complexity of the whole framework is thus $O(N^2L^2 + L^3D)$. Since the dependence analysis itself requires a lower bound complexity $O(N^2 * L^2)$, the whole framework has a complexity comparable to that of the dependence analysis required by every loop optimizing compiler. The framework is thus quite efficient and is eligible to be incorporated into production compilers.

7 Experimental Results

To evaluate both the effectiveness and efficiency of the dependence hoisting transformation framework introduced in this paper, we have implemented the framework as a Fortran source-to-source translator on top of the D System, an experimental compiler infrastructure developed at Rice university. We have applied the translator to optimize a collection of benchmarks. This section presents the performance measurements from optimizing these benchmarks.

We present the performance measurements for two classes of benchmarks: four linear algebra kernels — Cholesky, QR, LU factorization without pivoting, and LU factorization with partial pivoting — to verify dependence hoisting in blocking complex, non-perfect loop structures, and five large application benchmarks (shown in table 2) to verify our framework in combining loop interchange,fusion and blocking to optimize real-world

applications. Our transformation framework has successfully optimized both classes of benchmarks and has achieved significant performance improvements. Furthermore, the translator has spent limited compile-time in performing dependence hoisting analyses and transformations; in particular, the compile-time overhead of applying dependence hoisting proves to be comparable to that incurred by basic analysis and transformation techniques employed by standard optimizing compilers. These results indicate that the dependence hoisting technique is not only quite effective, it is also efficient enough for production compilers.

To make clear the effectiveness level of dependence hoisting, we further elaborate the significance of applying it to successfully block the four linear algebra kernels in our benchmark collection. These kernels are important linear algebra subroutines that are widely used in scientific computing. Furthermore, the loop nests within these kernels have been generally considered difficult to block automatically. Previous (and quite powerful) compiler techniques (including both unimodular transformation strategies [32, 34, 25, 7] and general loop transformation frameworks [21, 27, 1, 23]) have been able to automatically block only Cholesky and LU without pivoting. To our knowledge, few compiler techniques have completely automated the blocking of QR or LU with partial pivoting. Carr, Kennedy and Lehoucq [6, 8] hypothesized that no compiler could automatically produce the blocking of pivoting LU that is used in LAPACK [4] without specific commutativity information. This paper has not disproved this hypothesis; rather, it has succeeded in generating a substantially different blocking for pivoting LU without requiring such commutativity information. Surprisingly, our blocked version exhibits only minor performance differences (see section 7.2.1) when compared with the LAPACK version.

Being able to block both QR and pivoting LU indicates that with significantly less compile-time overhead, our translator can, in certain difficult cases, succeed where previous general transformation frameworks have failed. However, this observation does not generalize in a theoretical sense. Many general frameworks, which can be shown to be sufficiently powerful to succeed on a particular complex loop nest, may nevertheless fail to identify a valid transformation. One reason this might happen is that these systems require so much computational power that some solutions cannot be found due to technical constraints. Our approach is thus valuable in that it can handle difficult loop nests such as those in QR and pivoting LU and that it can do so with complexity similar to that of the inexpensive unimodular loop transformation techniques.

Before presenting the experimental results, Section 7.1 first briefly describes various versions of the bench-

Suite	Benchmark	Description	subroutine	No.lines
Spec95	Tomcatv	Mesh generation with Thompson solver	all	190
	Mgrid	Three dimensional multigrid solver	all	484
	Swim	Weather prediction program	all	429
ICASE	Erlebacher	Calculation of variable derivatives	all	554
NAS/ NPB2.3 - serial	SP	3D multi-partition algorithm	x_solve	223
			y_solve	216
			z_solve	220
			compute_rhs	417

Table 2: Application benchmarks used in evaluation

marks. Section 7.2 then presents performance measurements of the benchmarks on an SGI workstation. Finally, section 7.3 presents the compile-time overhead of optimizing these benchmarks.

7.1 Benchmarks

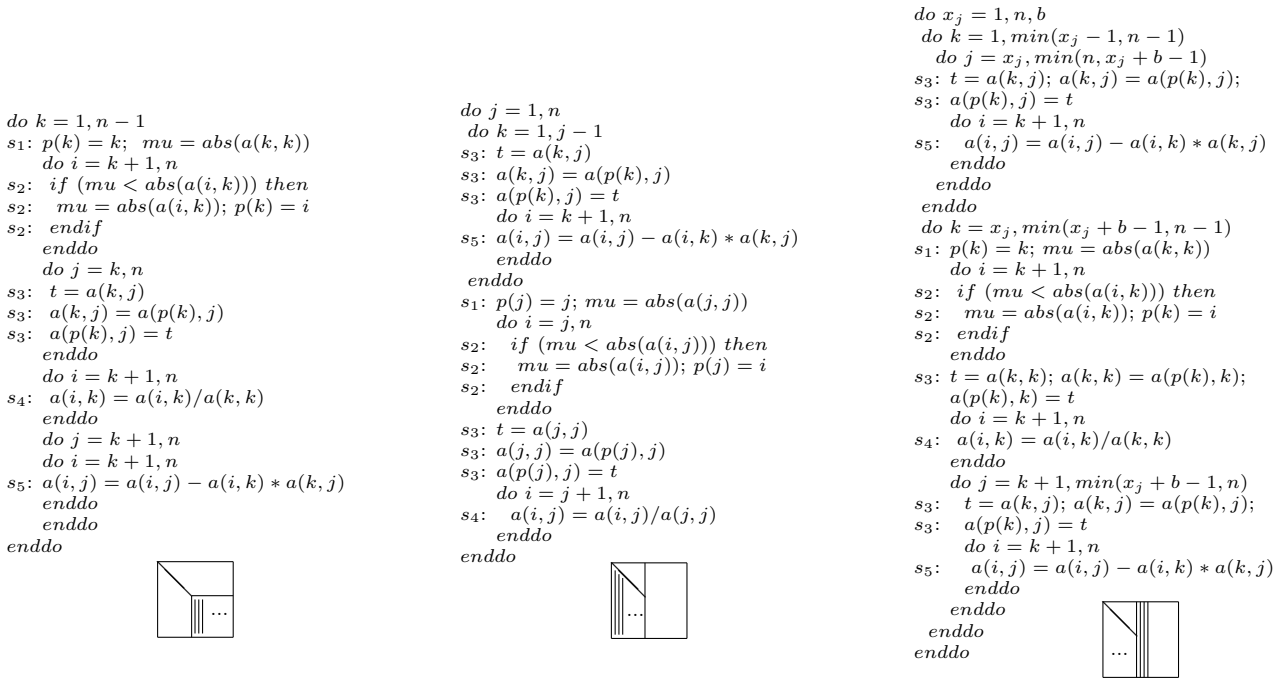
This section provides further information about the benchmarks used in our experiments. That is, the four numerical linear algebra kernels: *Cholesky*, *QR*, *LU factorization without pivoting*, and *LU factorization with partial pivoting*, and the five application benchmarks: *tomcatv*, *Erlebacher*, *mgrid*, *swim*, and *SP*. Table 2 describes each of the five application benchmarks.

The original versions of the four linear algebra kernels (*Cholesky*, *QR*, *LU factorization without pivoting*, and *LU factorization with partial pivoting*) are transcribed from the simple versions found in Golub and Van Loan [15]. The original versions of *LU* with and without pivoting are shown in Figure 13(a) and Figure 1(b) respectively. The original versions of *Cholesky* and *QR* are written similarly. The code for *QR* is based on Householder transformations [15].

The original versions of the large application benchmarks in Table 2 are downloaded from various benchmark suites, as shown in table 2. When given as input to the translators, these applications are used in their original forms from their respective benchmark suites with essentially no modification.

7.1.1 Blocking Linear Algebra Kernels

For the four linear algebra kernels, *Cholesky*, *QR*, *LU without pivoting*, and *LU with partial pivoting*, blocking is the principal optimization applied by our translator. For *Cholesky* and *non-pivoting LU*, both the row and column dimensions of the matrices are blocked; for *QR* and *pivoting LU*, only the column dimension is blocked



(a) original code (b) after dependence hoisting (c) after blocking

Figure 13: Blocking LU factorization with partial pivoting

(the row dimension cannot be blocked for the programs to be correct). For *non-pivoting LU*, the automatically blocked code by our translator is shown in Section 6.2.3. The blocking for *Cholesky* are quite similar to the one for *non-pivoting LU*. This section now briefly describes the blocking for *pivoting LU* by our translator, which has blocked *QR* similarly.

Figure 13(a) shows the original version of *pivoting LU*. This version is different from the one used in LAPACK BLAS [6] in that the j loop surrounding statement s_3 has iteration range "$j = k, n$" instead of "$j = 1, n$". To block the BLAS version, a preliminary step is needed to split this loop into two loops: "$j = 1, k - 1$" and "$j = k, n$". Although this step can be automated, it has not yet been implemented in our translator.

We block the pivoting LU code in Figure 13(a) using two computation slices: the first, $slice_k$, selects the outermost k loop as slicing loops for all the statements; the second, $slice_j$, selects the k loop for statements s_1,s_2 and s_4, but selects the j loops as slicing loops for s_3 and s_5. In Figure 13, the transformed code using $slice_j$ is shown in (b) and the blocked code using both slices is shown in (c). In the blocked code, the original $j(s_3)$ loop

```
      do 10 j=1,N
      do 10 i=1,N
       duz(i,j,1) = duz(i,j,1)*b(1)
10    continue
      do 20 k=2,N-1
      do 20 j=1,N
      do 20 i=1,N
       duz(i,j,k)=(duz(i,j,k)-a(k)*duz(i,j,k-1))*b(k)
20    continue
      do 30 j=1,N
      do 30 i=1,N
       tot(i,j) = 0.
30    continue
      do 40 k=1,N-1
      do 40 j=1,N
      do 40 i=1,N
       tot(i,j) = tot(i,j) + d(k)*duz(i,j,k)
40    continue
```

(a) original code

```
do j=1,N
  do i=1,N
    duz(i, j, 1) = duz(i, j, 1) * b(1)
    tot(i,j) = 0.
  enddo
  do k=1,N-1
    do i=1,N
      if (k .ge. 2) then
      duz(i, j, k) = (duz(i, j, k) - a(k) * duz(i, j, k - 1)) *b(k)
      endif
      tot(i,j) = tot(i,j) + d(k)*duz(i,j,k)
    enddo
  enddo
enddo
```

(b) after combined interchange and fusion

Figure 14: Multi-level fusion example: code fragment from Erlebacher

in (b) is split into two loops, "$j = k$" and "$j = k+1, n$", by step (2.1) of the dependence hoisting transformation algorithm in Figure 9. This blocked code operates on a single block of columns at a time and is similar to the blocked code of non-pivoting LU in Figure 11(c). This blocking strategy is based on the observation that, although the code dealing with selecting pivots in Figure 13(a) imposes bi-directional dependence constraints among rows of the input matrix, the dependence constraints among columns of the matrix have only one direction—from columns on the left to columns on the right. Therefore the factorization can be blocked in the column direction of the matrix.

7.1.2 Optimizing Application Benchmarks

For the five application benchmarks, *tomcatv*, *swim*, *mgrid*, *Erlebacher*, and *SP*, all three loop optimizations, interchange, fusion and blocking, are performed by the translator. However, because these benchmarks do not exhibit as many temporal data reuses as the linear algebra kernels, loop blocking is not as beneficial for them. As a result, blocking has improved the overall performance only for *mgrid*.

To illustrate a combined interchange and multi-fusion effect achieved by the dependence hoisting framework, Figure 14 shows both the original and optimized versions of a fragment in subroutine *tridvpk* from *Erlebacher* (a benchmark for computing partial derivatives). The original code in Figure 14(a) contains four loop nest, denoted as N_{10}, N_{20}, N_{30} and N_{40} respectively. The optimized code after applying dependence hoisting is shown in (b).

All the loops in Figure 14(a) can be legally shifted to the outermost loop level. Therefore when optimizing

this code, our framework has constructed a computation slice for each loop, and similarly a slice nest for each loop nest in the original code. The framework then applies the typed-fusion algorithm to these constructed slice nests. As multiple slices in each slice nest are considered for fusion simultaneously, the framework can automatically fuse loops initially at different loop levels. For example, as all the j loops in (a) can be fused into a single loop, the fusion algorithm fuses all the corresponding j slices and then uses the fused slice to perform a dependence hoisting transformation, which generates the outermost j loop in (b). All the i loops in (a) can also be fused into a single loop; however, because these i loops carry spatial reuses and should stay innermost, the framework decides not to fuse all the i loops after profitability analysis.

7.2 Performance Measurements of Benchmarks

This section presents performance measurements of the benchmarks. The performance results are measured on an SGI workstation with a 195 MHz R10000 processor, 256MB main memory, separate 32KB first-level instruction and data caches (L1), and a unified 1MB second-level cache (L2). Both caches are two-way set-associative. The cache line size is 32 bytes for L1 and 128 bytes for L2. For each benchmark, the SGI's *perfex* tool (which is based on two hardware counters) is used to count the *total* number of cycles, and L1, L2 and TLB misses.

All the benchmarks (including their original versions, automatically optimized versions and manually optimized versions) were compiled using the SGI F77 compiler with "-O2" option. which directs the SGI compiler to turn on extensive optimizations. The optimizations at this level are generally conservative and beneficial, and they do not perform aggressive transformations such as global fusion/distribution or loop interchange. All the versions are optimized at the same level by the SGI compiler. Each measurement is repeated 5 or more times and the average result across these runs is presented. The variations across runs are very small (within 1%).

7.2.1 Performance of Linear Algebra Kernels

To show the power of dependence hoisting in blocking complex loop structures, this section presents the performance measurements of four linear algebra kernels, Cholesky, QR, LU without pivoting, and LU with partial pivoting. For each kernel, the performance of the auto-blocked code by our translator is compared with that of the original code and that of the *out-of-the-box* LAPACK subroutine except for LU without pivoting (there is no

such LAPACK entry). For non-pivoting LU, the performance of the auto-blocked code is compared with that of a version blocked by hand following the LAPACK blocking strategy for LU with pivoting [6].

The objective in comparing with LAPACK is to show how close the auto-blocked versions can get to the best hand-coded versions of the same kernels — LAPACK is chosen because it has been developed by professional algorithm designers over a period of years. In some cases, the developers even applied algorithmic changes that are not available to compilers because these changes violate the dependence restrictions. Our translator has achieved performance improvements comparable to those achieved by LAPACK. This fact indicates that the translator is very effective in optimizing complex loops of the kinds found in linear algebra kernels.

Note that the comparison with LAPACK is not attempting to compete auto-translated versions with the existing hand-tuned implementations. Instead, the naive versions of the kernels are used to demonstrate the power of our translator. If the automatic blocking strategy can succeed in optimizing these kernels to a level of performance comparable to LAPACK, it can also be successful on a wide variety of similarly challenging loop nests.

For each auto-translated version, different block sizes were tested and the result from using the best block size is presented. Thus, the results for auto-blocked versions include the improvements due to an auto-tuning step similar to (but less powerful than) that used by the ATLAS system to tune the BLAS for a new architecture [30]. This is fair because the numbers reported for LAPACK in Figure 15 are based on BLAS versions with tuning parameters selected by ATLAS for the SGI workstation.

Figure 15 presents the performance results of the linear algebra kernels. For each kernel, this figure presents the measurements using two matrix sizes: a moderate size (500^2) and an enhanced size (1000^2). Each set of measurements is normalized to the performance of the original version. The following discussion denotes each blocked version by our translator as a *sliced version* and denotes each version blocked by LAPACK as an LAPACK version.

All the sliced versions are able to perform much better than the original versions because of better locality. The performance improvements are shown uniformly in the cycle count and L2 cache miss graphs. The sliced versions for *Cholesky* and *non-pivoting LU* also manifest improvements in L1 cache by more than a factor of 10. Here because our translator has blocked both the row and column dimensions of the matrices, the working sets of

- Versions: o—original; s—blocked by our translator; L—blocked by LAPACK.
- Benchmarks:

Cholesky:	chl1—500^2 matrix;	chl2—1000^2 matrix;
Non-pivoting LU:	lu1—500^2 matrix;	lu2—1000^2 matrix;
Pivoting LU:	lup1—500^2 matrix;	lup2—1000^2 matrix;
QR:	qr1—500^2 matrix;	qr2—1000^2 matrix.

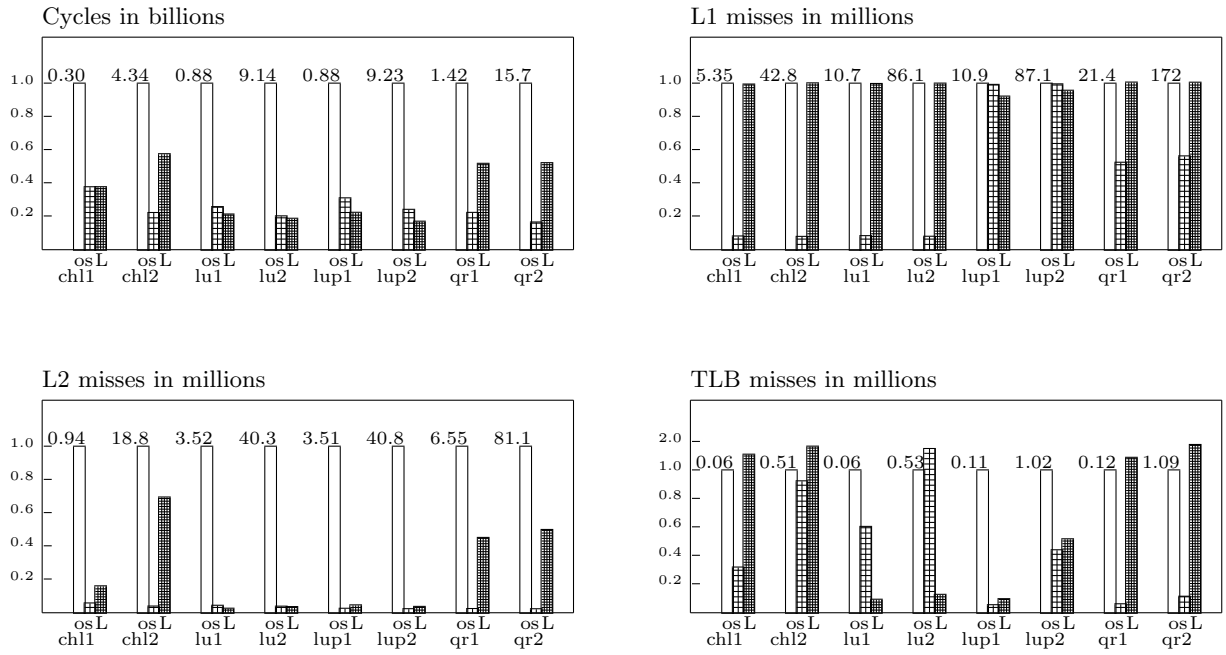


Figure 15: Results from blocking linear algebra kernels

these versions are small enough to fit in L1 cache. However, these two versions also show worse TLB performance improvements due to accessing data in large strides. In contrast, the sliced versions of *QR* and *pivoting LU* have only the column dimension blocked and thus have large, contiguous computation blocks. Consequently, these two versions have better TLB performance improvements (due to accessing data with small strides) but worse L1 cache improvements (due to large block sizes). In particular, the L1 cache performance improvements for the auto-blocked versions are 50% improvement for *QR* and no improvement for *pivoting LU*.

When compared with the LAPACK versions, the sliced version of *Cholesky* achieves almost identical overall performance using a 500^2 matrix and achieves much better performance using a 1000^2 matrix; the sliced version of *QR* achieves better performance than the LAPACK version using both matrix sizes; and the sliced versions of *non-pivoting* and *pivoting LU* achieve a performance level comparable to yet slightly worse than the LAPACK

- Versions: o—original; f—optimized with interchange+fusion; b—optimized with interchange+fusion+blocking.
- Benchmarks:

tomcatv:	tcat1—513^2 matrix, 750 iterations;	tcat2—1025^2 matrix, 750 iterations;
mgrid:	mg1—64^3 grid, 1000 iterations;	mg2—256^3 grid, 1 iteration;
Erlebacher:	Erle1—129^3 grid;	Erle2—256^3 grid;
swim:	sw—512^2 matrix, 900 iterations;	
SP:	SP—102^3 grid (class B), 3 iterations.	

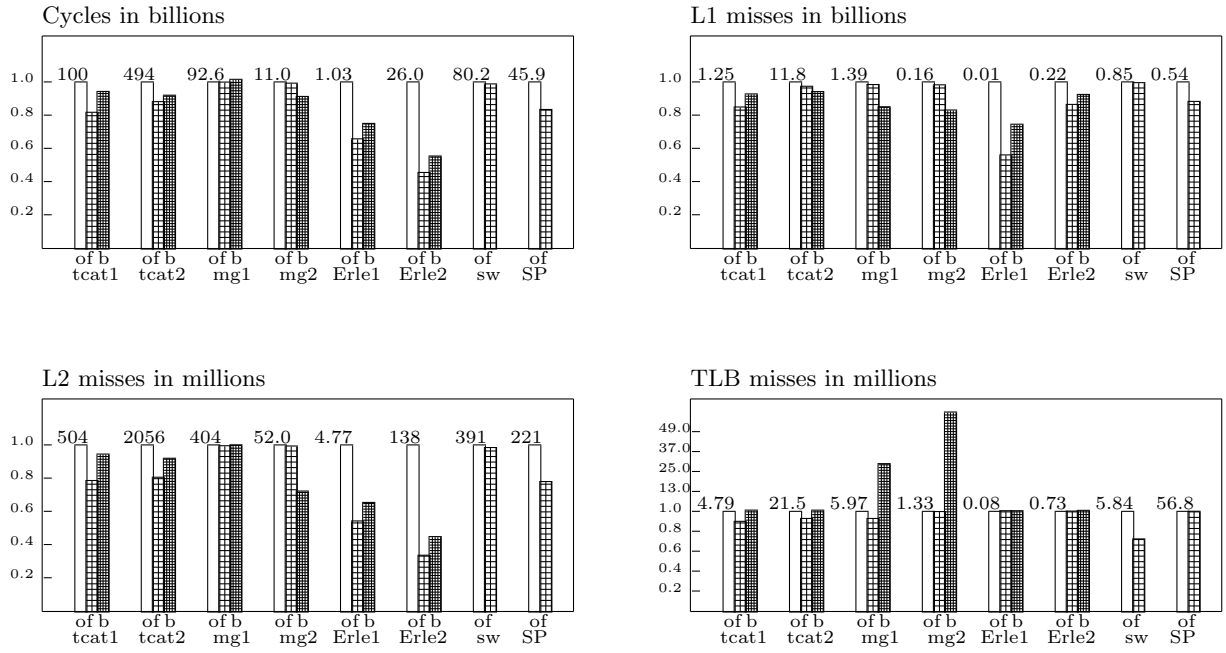


Figure 16: Results from optimizing application benchmarks

versions. For *pivoting LU*, both the sliced and LAPACK versions have only the column dimension of the matrix blocked (the row dimension cannot be blocked in order for the program to be correct), and the SGI workstation favors the loop ordering in the LAPACK version. For *non-pivoting LU*, the penalty of higher TLB misses for the sliced version negates its better cache performance. Further optimizations that reorganize data layout [28, 11] are necessary to improve this situation.

7.2.2 Performance of Application Benchmarks

To illustrate the effectiveness of our translator in achieving loop interchange, fusion and blocking optimizations for real-world applications, this section presents the performance measurements for the five large application benchmarks, shown in Table 2.

Figure 16 shows the performance results of these benchmarks. Here multi-level loop fusion is the principal beneficial optimization for all the benchmarks except *mgrid*. Loop blocking is applied to only three of the five benchmarks because it is guaranteed to be non-profitable for the other benchmarks (*swim* and *SP*).

For each application that may benefit from both loop fusion and blocking, we provide two auto-optimized versions: one version is optimized by loop interchange and fusion only, and the other is optimized with all three transformations. Because our translator refrains from fusing loops when fusion inhibits blocking, by turning off blocking optimization in our translator, we explicitly evaluate the tradeoff between applying fusion and blocking. For each of the benchmarks optimized by blocking, Figure 16 also presents measurements using two different data sizes, a moderate size and an enhanced size, to illustrate the varying effect of blocking. Because the actual execution time of *mgrid* and *SP* using enhanced grid sizes (256^3 for *mgrid* and class B for *SP*) has become unreasonably prolonged, only a single time step (one iteration for *mgrid* and three iterations for *SP*) is used for measuring both benchmarks.

For all the application benchmarks, loop fusion has improved both the cycle counts and the performance at all cache levels. The improvement is a minor $1-2\%$ for *mgrid* and *swim*, but is much higher for other benchmarks: $12-18\%$ for *tomcatv*, 17% for SP, and $34-55\%$ for *Erlebacher*. The improvement for *tomcatv* comes from fusing two loop nests inside the outermost time-step loop. The improvement for *SP* comes mostly from fusing the loop nests in subroutine *compute_rhs*, which yields an extra 12% improvement over optimizing the subroutines *x_solve*, *y_solve* and z_solve alone. The improvement for *Erlebacher* comes from aggressive combined interchange and fusion optimization in several subroutines. For *mgrid* and *swim*, the fusion optimization does not make a major difference because most of the subroutines in these applications contain only one or two loop nests, and fusing these loop nests does not provide much benefit. Subroutine inlining and aggressive index-set splitting [11] may be able to further improve the performance of these applications.

From Figure 16, loop blocking provides further improvement after loop fusion only for *mgrid* using the 256^3 grid. Here blocking achieves a further 8% overall improvement after fusion due to the significant reduction of L1 and L2 cache misses, as shown in the L1 and L2 miss graphs of *mgrid*. However, due to accessing data in large strides, blocking also immensely degrades the TLB performance of *mgrid*. Due to similar reasons, loop blocking provides no further overall improvement after fusion for either *mgrid* with a smaller grid size, or *tomcatv* and

Erlebacher using both data sizes. For *tomcatv* and *Erlebacher*, unless a single row of the matrix (or grid) exceeds the cache size, loop blocking cannot provide any extra data reuses after fusion. For *tomcatv* with a 1025^2 matrix, loop blocking provides a minor improvement after fusion in L1 cache performance, but this effect is negated by the performance loss in L2 and TLB performance. For *tomcatv* with a 512^2 matrix and for *Erlebacher* with both grid sizes, all the data accesses can still fit in cache, so blocking degrades performance at all the cache levels.

Comparing the performance measurements using different data sizes for each benchmark, we see that as the data size increases, the performance improvements from both fusion and blocking optimizations move from the L1 cache to the L2 cache, and its overall impact on performance improvements vary. For *mgrid* and *Erlebacher*, better performances are achieved, but for *tomcatv*, the opposite conclusion holds. Note that larger data sizes severely degrades the the TLB performance after loop blocking (shown in the TLB performance of auto-blocked *mgrid*) due to increased data accessing strides, but it does not affect the TLB performance if only loop fusion is applied (shown in the TLB performance numbers for versions optimized by interchange and fusion only).

7.3 Compile Time Evaluation

This section presents the compile-time measurements of our translator when optimizing the benchmarks. The translator is written in C++ and is itself compiled with "-g" option (the lowest optimization level by the C++ compiler). The compile-time overhead is measured as the elapsed time of the compilation on a SUN Sparc workstation with 336MHz processors. We present both the elapsed time of the whole compilation and the time spent in the actual dependence hoisting analyses and transformations. Each measurement is repeated 15 times and the average value across runs is presented (the variation is within 2%).

Table 3 presents the compile-time measurements of optimizing each benchmark using the translator. Here the benchmarks are listed in increasing order of their code sizes, which are computed using the following formula.

$$\sum_{\forall s \in C} loop\_level(s) \tag{21}$$

Here for each non-loop statement s in a benchmark C, $loop\_level(s)$ is the number of loops surrounding s. The code size for each benchmark is thus computed as the number of different loop-statement pairs in the benchmark. Because benchmark SP has four subroutines x_solve, y_solve, z_solve and $compute\_rhs$, each subroutine

Benchmark	Number of subroutines	Number of stmt-loop pairs	Compile time(seconds)	
			t_{all}	t_{slice}
LU non-pivoting	1	5	0.521	0.283
Cholesky	1	6	0.633	0.411
LU pivoting	1	15	1.344	0.964
QR	1	17	1.48	1.053
mgrid	9	71	8.472	1.651
swim	6	82	6.747	1.497
tomcatv	1	95	5.719	2.580
Erlebacher	8	99	11.648	2.621
z_solve	1	149	16.595	8.113
y_solve	1	149	16.63	8.113
x_solve	1	149	17.049	8.578
compute_rhs	1	161	24.046	7.979

Table 3: Compile time of optimizing benchmarks

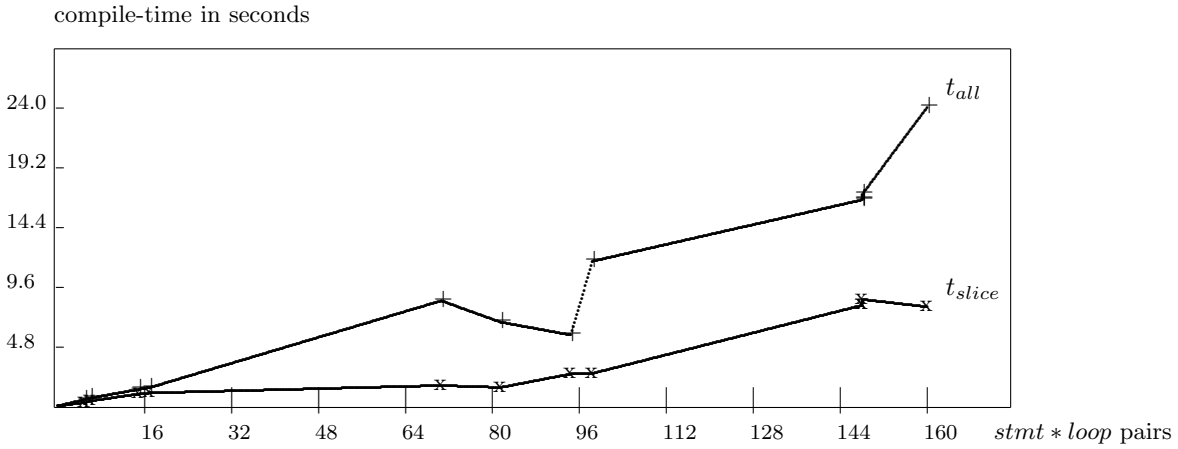


Figure 17: Correlation between compile-time and code-sizes of the benchmarks

contained in a separate file, the translator has optimized these subroutines separately, and we measured their compilation times as if they are individual benchmarks.

Table 3 presents two compile-time measurements for each benchmark: one marked as t_{all} and the other marked as t_{slice}. The value t_{all} refers to the elapsed time of the whole compilation for optimizing a benchmark. The value t_{slice} refers to the time spent in the actual dependence hoisting analyses and transformations; that is, t_{slice} is equal to t_{all} subtracting the time spent in program I/O, parsing, code generation, and basic analyses such as control-flow and dependence analysis. Figure 17 shows the correlation between the compile time measurements and the code sizes of the benchmarks.

From the measurements of t_{slice} in Table 3, the dependence hoisting technique is quite efficient. The time

spent in optimizing each benchmark stays less than three seconds except when the benchmark contains more than 100 loop-statement pairs. The time spent in optimizing each loop-statement pair is roughly 0.02 seconds for *mgrid*, *swim*, *tomcatv* and *erlebacher* and is roughly 0.05-0.07 seconds for the other benchmarks. This efficacy of dependence hoisting (compile-time overhead per loop-statement pair) is further plotted in Figure 17, which shows that the compile-time overhead of dependence hoisting increases roughly proportionally with the code sizes of the benchmarks. The variation of this efficacy for different benchmarks is caused by the different sizes of dependence graphs in these benchmarks.

By comparing t_{all} and t_{slice} in Table 3, we see that the overall compile-time of our translator is 1.4-1.8 times t_{slice} for the four linear algebra kernels and is 2-5 times t_{slice} for the other benchmarks. As the code size increases, t_{slice} stays less than half of t_{all} and increases with the code size at a similar rate as that of the rest of the compile-time overhead, which is incurred by standard compilation phases that are included in standard optimizing compilers. The dependence hoisting analysis and transformation thus will not become the bottleneck of a production optimizing compiler as the code sizes of the benchmarks increase. It is therefore efficient enough to be incorporated into these production compilers.

8 Related Work

Many compiler techniques have been developed to improve the performance of memory hierarchies. The most popular is a set of unimodular and single loop transformations, such as loop blocking, fusion, distribution, interchange, skewing and index-set splitting [32, 22, 25, 10, 14, 34, 7]. These techniques are inexpensive and are widely used in production compilers to optimize applications with simple loop structures both for locality and for parallelism. However, these techniques are not effective enough when transforming complex,non-perfectly nested loop structures that cannot be translated into sequences of perfect loop nests. Wolf and Lam [32] proposed an uniform algorithm to select compound sequences of unimodular loop transformations for non-perfect loop nests. This algorithm is still limited by the original loop structures of programs.

The dependence hoisting technique in this paper extends these traditional techniques to effectively transform complex loop nests independent of their original nesting structure. Although it does not incorporate the entire

solution space of loop transformations, our technique has demonstrated high effectiveness by blocking some of most challenging benchmarks.

Several general loop transformation frameworks [21, 27, 1, 23, 31] are theoretically more powerful but are also much more expensive than the dependence hoisting technique introduced in this paper. These general frameworks typically adopt a mathematical formulation of program dependences and loop transformations. They first compute a mapping from the iteration spaces of statements into some unified space. The unified space is then considered for transformation. Finally, a new program is constructed by mapping the selected transformations of the unified space onto the iteration spaces of statements. The computation of these mappings is expensive and generally requires special integer programming tools such as the Omega library [16]. Because of their high cost, these frameworks are rarely used in commercial compilers. In contrast, we seek simpler yet also highly effective solutions with a much lower compile-time overhead.

The general frameworks discussed above vary in their efficiency and effectiveness. In particular, Pugh [27] proposed a framework that finds a schedule for each statement describing the moment each statement instance will be executed. This technique requires an expensive step to find a feasible mapping from iteration spaces of statements into instance execution *time*. Kodukula, Ahmed and Pingali [21] proposed an approach called *data shackling*, in which a tiling transformation on a loop nest is described in terms of a tiling of key arrays in the loop nest. Here a mapping from the iteration spaces of statements into the *data* spaces must be computed. Lim, Cheong and Lam [23] proposed a technique called *affine partition*, which maps instances of instructions into the *time* or *processor* space via an affine expression in terms of the loop index values of their surrounding loops. Finally, Ahmed, Mateev and Pingali [1] proposed an approach that embeds the iteration spaces of statements into a *product space*. The product space is then transformed to enhance locality. In this approach, a mapping from iteration spaces of statements into the *product space* must be computed.

The dependence hoisting transformation framework presented in this paper is a compromise that trades a small amount of generality for substantial gains of efficiency. Our framework does not manipulate any mathematically formulated symbolic space. Instead, it relates the iterations of loops directly using a matrix of dependence directions and distances. Although it is less general than the above mathematically formulated frameworks, our framework is powerful enough for a large class of real-world applications and is much more efficient. This

framework can be combined with traditional transformation systems for simple loop nests and, because it is inexpensive, is suitable for inclusion in commercial production compilers.

The loop model in this paper is similar to the one adopted by Ahmed, Mateev and Pingali [1]. They represent the same loop surrounding different statements as different loops and use a product space to incorporate all the extra loop dimensions. This work uses far fewer extra loop dimensions. Our framework temporarily adds one extra loop at each dependence hoisting transformation and then removes the extra dimension immediately.

Pugh and Rosser was the first to propose using transitive dependence information for transforming arbitrarily nested loops [29, 17]. They represent transitive dependences using integer set mappings and apply an enhanced Floyd-Warshall algorithm to summarize the complete dependence paths between statements. Their dependence representation and transitive analysis algorithm are quite expensive. This paper has developed a different dependence representation and transitive analysis algorithm, both of which are much more efficient. These improvements make transitive dependence analysis fast enough for incorporation in production compilers.

Finally, the transformation framework in this paper can be further extended to integrate many other compiler techniques both for optimizing memory performance and for optimizing parallel performance. These techniques include automatic selection of blocking factors [10, 22, 26], heuristics for loop fusion [20, 18], multi-level memory hierarchy management [9], and data layout rearrangement transformations [28].

9 Conclusion

This paper extends previous unimodular loop transformation techniques to effectively optimize complex, non-perfect loop structures. We have introduced a novel loop transformation technique, dependence hoisting, that facilitates the fusion and interchange of arbitrarily nested loops. We have also extended the traditional dependence model to determine the safety of dependence hoisting and have presented a transformation framework that integrates dependence hoisting with traditional unimodular and single loop transformations, such as loop interchange, fusion, strip-mining, reversal and index-set splitting, to optimize arbitrary loop structures for memory hierarchy performance. Our transformation framework is comparable in complexity to the traditional unimodular loop transformation systems and is in practice also comparable in power to the more general transformation

frameworks that utilize expensive integer-set mapping techniques.

References

[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proceedings of the 2000 ACM International Conference on Supercomputing*, Santa Fe, New Mexico, May 2000.

[2] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, June 1984.

[3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, San Francisco, October 2001.

[4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide.* The Society for Industrial and Applied Mathematics, 1999.

[5] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, 1988.

[6] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing*, Minneapolis, Nov. 1992.

[7] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[8] S. Carr and R. Lehoucq. Compiler blockability of dense matrix factorizations. *ACM Trans. Math. Softw.*, 23(3), 1997.

[9] L. Carter, J. Ferrante, and S. F. Hummel. Hierarchical Tiling for Improved Superscalar Performance. In *Proc. 9th International Parallel Processing Symposium*, Santa Barbara, CA, Apr. 1995.

[10] S. Coleman and K. S. McKinley. Tile size selection using cache organization. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[11] C. Ding. *Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse*. PhD thesis, Rice University, 2000.

[12] J. Dongarra, J. Bunch, C. Moler, and G.Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1979.

[13] J. J. Dongarra, F. G. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):91–112, Jan. 1984.

[14] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, Oct. 1988.

[15] G. H. Golub and C. F. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, second edition, 1989.

[16] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library Interface Guide. Technical report, Dept. of Computer Science, Univ. of Maryland, College Park, Apr. 1996.

[17] W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive Closure Of Infinite Graphs And Its Applications. *International Journal of Parallel Programming*, 24(6), Dec 1996.

[18] K. Kennedy. Fast greedy weighted fusion. In *Proceedings of the International Conference on Supercomputing*, Santa Fe, NM, May 2000.

[19] K. Kennedy and K. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the ACM International Conference on Supercomputing*, July 1992.

[20] K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report TR93-208, Dept. of Computer Science, Rice University, Aug. 1993. (also available as CRPC-TR94370).

[21] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 1997.

[22] M. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Santa Clara, Apr. 1991.

[23] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[24] N. Manjikian and T. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):193–209, Feb. 1997.

[25] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[26] N. Mitchell, L. Carter, J. Ferrante, and K. Hgstedt. Quantifying the multi-level nature of tiling interactions. In *10th International Workshop on Languages and Compilers for Parallel Computing*, August 1997.

[27] W. Pugh. Uniform techniques for loop optimization. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, June 1991.

[28] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[29] E. J. Rosser. *Fine Grained Analysis Of Array Computations*. PhD thesis, Dept. of Computer Science, University of Maryland, Sep 1998.

[30] R. Whaley and J. Dongarra. Automatically tuned linear algebra software(atlas). In *Proceedings of Supercomputing '89*, 1989.

[31] William Pugh and Evan Rosser. Iteration Space Slicing For Locality. In *LCPC 99*, July 1999.

[32] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, June 1991.

[33] M. J. Wolfe. Advanced loop interchanging. In *Proceedings of the 1986 International Conference on Parallel Processing*, St. Charles, IL, Aug. 1986.

[34] M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, Reno, Nov. 1989.

[35] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, 1989.

[36] Q. Yi, V. Adve, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, British Columbia, Canada, June 2000.