Improving Memory Hierarchy Performance Through Combined Loop Interchange and Multi-Level Fusion

Qing Yi Ken Kennedy

Computer Science Department, Rice University MS-132

Houston, TX 77005

Abstract

Because of the increasing gap between the speeds of processors and main memories, compilers must enhance the locality of applications to achieve high performance. Loop fusion enhances locality by fusing loops that access similar sets of data. Typically, it is applied to loops at the same level after *loop interchange*, which first attains the best nesting order for each local loop nest. However, since loop interchange cannot foresee the overall optimization effect, it often selects the wrong loops to be placed outermost for fusion, achieving sub-optimal performance globally. Building on traditional *unimodular* transformations on perfectly nested loops, we present a novel transformation, *dependence hoisting*, that effectively combines interchange and fusion for arbitrarily nested loops. We present techniques to simultaneously interchange and fuse loops at multiple levels. By evaluating the compound optimization effect beforehand, we have achieved better performance than that was possible by previous techniques, which apply interchange and fusion separately.

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 insert one or more levels of cache between the processor and memory, resulting in a 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, compliers must optimize the locality of applications so that data fetched into caches are reused before being displaced.

Loop fusion improves the memory hierarchy performance by fusing loops that access similar sets of memory locations. After fusion, the accesses to these locations are brought closer together and thus can be reused inside the fused loop. Traditionally, loop fusion can be applied only to a sequence of loops at the outermost position of a code segment. Although loop interchange can be applied before fusion to bring the desired loops outermost, loop interchange can also place the wrong loops at the outermost position for fusion, because it cannot foresee the overall effect of fusion transformations.

To illustrate this situation, we show a code segment from *Erlebacher*, a benchmark application from ICASE, in Figure 1. The original code in (a) contains four loop nests, marked as nest 40, 50, 60 and 70 respectively. The code in (b) is transformed from (a) by applying loop interchange. Here the j loop in nest 70 is interchanged with the k loop.

Figure 1(c) shows the transformed code from (b) after applying traditional loop fusion. Here the j loops from the original nests 50,60 and 70 are fused into a single loop, and so are the i loops in the original nests 50 and 60. Note that nest 40 cannot participate in the fusion because its outermost k loop cannot be legally fused with the other j loops at the outermost level. If the compiler had interchanged the j and k loops in nest 40, the fusion would have been possible, and the transformed code in (d) would have been attained. The code in (d) has better locality than (c) because here the accesses to array $tot(i, j)(i = 1, N)$ in nest 40 are more likely to be reused.

Note that the best nesting order of nest 40 depends on the value of N and the cache sizes of the machine. Without such knowledge, it is unclear which of the j and k loops should be placed outermost. However, if a compiler can foresee the extra reuses made possible by fusing nest 40 with the other loop nests, it can easily determine that placing the j loop outermost is more beneficial. It is therefore necessary for the compiler to

consider the compound effect of both loop interchange and fusion when arranging loop nesting orders.

This paper presents techniques to combine loop interchange and fusion, and to evaluate the compound effect of both transformations when applying them to improve memory hierarchy performance. Our strategies can easily achieve the transformed code in Figure 1(d) without depending on the output of a previous loop interchange step. We have applied these strategies to optimize four benchmark applications, *tomcatv* (a mesh generation code from SPEC95), *Erlebacher* (a partial differentiation code from ICASE), *SP* (a 3D multi-partition code from NAS), and *twostages* (a kernel subroutine from a weather prediction system). We compare the performance of optimizing these applications with different interchange and fusion strategies, providing evidence that to achieve optimal memory performance, one must consider the impact of all optimizations collectively, which is a hard task to request from any programmer and thus should be left to compiler optimizations.

As a second contribution, we also present a novel transformation, *dependence hoisting*, that facilitate a combined loop interchange and fusion transformation on arbitrarily nested loops. Because it is independent of the original nesting structure, this transformation can facilitate the interchange and fusion of complex loop structures that were previously considered impossible to optimize automatically. We illustrate this transformation using a code for LU factorization without pivoting in Section 3. We also present the performance results of applying this transformation to optimize four linear algebra kernels, Cholesky, QR, LU factorization without pivoting, and LU with partial pivoting. All of these kernels contain complex loop nests, and the automatic interchange of some nests have not been achieved previously by a compiler.

To elaborate the framework, Section 2 first introduces an extended dependence model for determining the safety of transforming arbitrarily nested loops. Section 3 introduces a novel transformation, dependence hoisting, to facilitate the combined loop interchange and fusion. Section 4 describes algorithms that systematically perform the combined loop interchange and multi-level fusion optimization for better memory hierarchy performance. Section 5 presents experimental results. Section 6 summarizes related work. Finally, conclusions are drawn in Section 7.

2 Extended Dependence Model

In traditional loop transformation systems, each dependence from a statement s_1 to s_2 is associated with a vector that defines a direction or distance relation for each common loop surrounding both s_1 and s_2. To model the safety of transforming arbitrarily nested loops, We extend the traditional model with a new dependence representation, *Extended Direction Matrix(EDM)*, which defines direction and distance relations between iterations of non-common loops as well. Our model also summarizes the transitive dependence information between statements and then uses the summarized information to determine the safety of transforming non-perfectly nested loops.

In the following, Section 2.1 first introduces some notations. Section 2.2 describes the EDM representation of dependences and transitive dependences. Section 2.3 then summarizes how to use our extended dependence model to determine the safety of applying loop interchange and fusion to arbitrarily nested loops.

2.1 Notations and Definitions

To fuse loops without being limited by their original nesting structure, we adopt a loop notation similar to that by Ahmed, Mateev and Pingali [1]. We use $\ell(s)$ to denote a loop ℓ surrounding some statement s. This notation enables us to treat loop $\ell(s)$ as different from loop $\ell(s')$ $(s' \neq s)$ and thus to treat each loop ℓ surrounding multiple statements as potentially distributable. Dependence analysis will later inform us whether or not each loop can 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 $\ell_i(s)(i = 1, ..., m)$, we denote its iteration index variable as $Ivar(\ell_i(s))$ and its iteration range as $Range(\ell_i(s))$. Each value I of $Ivar(\ell_i(s))$ defines a set of iteration instances of s, a set that can be expressed as $Range(\ell_1) \times ... \times Range(\ell_{i-1}) \times I \times ... \times Range(\ell_m)$. We denote this iteration set 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 loops inside other control structures such as conditional branches. We adopt this strategy to simplify the technical presentation of this paper. To optimize real-world applications, preparative transformations such as "if conversion" [2] must be incorporated to remove the non-loop control structures in between loops. These preparative transformations are out of the scope of this paper and will not

be discussed further.

In this paper, we discuss loop transformations only at the outermost loop level of a code segment. To apply transformations at deeper loop levels, we hierarchically consider the code segment at each loop level and apply the same algorithms at the outermost level of that code segment, guaranteeing that no generality is sacrificed.

2.2 Dependences and Transitive Dependences

We use $d(s_x, s_y)$ to denote the set of dependence edges from statement s_x to s_y in the dependence graph. Suppose that s_x and s_y are surrounded by m_x loops, $(\ell_{x1}, \ell_{x2}, \ldots \ell_{xm_x})$, and m_y loops, $(\ell_{y1}, \ell_{y2}, \ldots \ell_{ym_y})$, respectively. Each dependence edge from s_x to s_y is represented using an $m_x \times m_y$ matrix $D_{m_x m_y}$ which we call an *Extended Direction Matrix (EDM)*. Each entry $D[i, j]$ $(1 \leq i \leq m_x,\ 1 \leq j \leq m_y)$ in the matrix specifies a dependence condition between loops $\ell_{xi}(s_x)$ and $\ell_{yj}(s_y)$. The dependence represented by D satisfies the conjunction of all these conditions. This EDM representation extends traditional dependence vectors [2, 26] by computing a relation between two arbitrary loops $\ell_x(s_x)$ and $\ell_y(s_y)$ even if $\ell_x \neq \ell_y$. The extra information is important for resolving safety of loop transformations independent of the original loop structure.

Given a dependence EDM D from statement s_x to s_y, we use $D(\ell_{xi}, \ell_{yj})$ to denote the dependence condition in D between loops $\ell_{xi}(s_x)$ and $\ell_{yj}(s_y)$. 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. We use the notation $Dir(D(\ell_{xi}, \ell_{yj}))$ to denote the *dependence direction* ("=", "$\leq$", "$\geq$" or "$*$") of the condition and use $Align(D(\ell_{xi}, \ell_{yj}))$ to denote the alignment factor of the condition. Both the dependence directions and alignment factors can be computed using traditional dependence analysis techniques [2, 26, 3].

We use the notation $td(s_x, s_y)$ to denote the set of transitive dependence edges from s_x to s_y. This set includes all the dependence paths from s_x to s_y in the dependence graph. Each transitive dependence edge in $td(s_x, s_y)$ has the same EDM representation as those of the dependence edges in $d(s_x, s_y)$. Because they summarize the complete dependence information between statements, transitive dependence edges can be used directly to determine the legality of program transformations, as shown in Section 2.3. To compute these

transitive dependence edges, we perform transitive analysis on the dependence graph using the algorithm by Yi, Adve and Kennedy [27][1]

2.3 Transformation Safety Analysis

Using transitive dependence information, this section resolves the legality of fusing arbitrarily nested loops at the outermost loop level of a given code segment; that is, the fused loop is placed at the outermost loop level. In particular, we determine 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.

To decide whether or not a loop $\ell(s)$ can be legally shifted to the outermost loop level, we examine $td(s, s)$ (the transitive dependence set from statement s to itself). We conclude that the shifting is legal if the following equation holds:

$$\forall D \in td(s,s),\ D(\ell,\ell) = \text{“}=n\text{”}\ or\ \text{“}\leq n\text{”},\ where\ n \leq 0. \tag{1}$$

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 $\ell(s)$ 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:

$$\begin{aligned} &\forall D \in td(s_y,s_x), \quad Dir(D(\ell_y,\ell_x)) = \text{“}=\text{”}\ or\ \text{“}\leq\text{”} \quad and \\ &\forall D \in td(s_x,s_y), \quad Dir(D(\ell_x,\ell_y)) = \text{“}=\text{”}\ or\ \text{“}\leq\text{”}. \end{aligned} \tag{2}$$

If Equation (2) holds, we can fuse loops $\ell_x(s_x)$ and $\ell_y(s_y)$ into a single loop $\ell_f(s_x, s_y)$ s.t.

$$Ivar(\ell_f) = Ivar(\ell_x) = Ivar(\ell_y) + align, \tag{3}$$

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

[1] In [27], we used an integer programming tool, the Omega Library [12], to facilitate recursion transformation. However, the transitive analysis algorithm in [27] is independent of specific dependence representations and does not use the Omega Library.

following equation:

$$
\begin{aligned}
a_y \leq\ & align \leq -a_x, where \\
& a_x = Max\{\ Align(D(\ell_y, \ell_x)),\ \ \forall D \in td(s_y, s_x)\ \}, \\
& a_y = Max\{\ Align(D(\ell_x, \ell_y)),\ \ \forall D \in td(s_x, s_y)\ \}.
\end{aligned} \tag{4}
$$

From Equation (3), 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 (2) and (4), 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 (4), 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.

3 Dependence Hoisting

This section introduces a novel technique, dependence hoisting, that facilitate a combined loop interchange and fusion on a group of arbitrarily nested loops. This technique has a similar complexity as that of the traditional loop fusion/distribution transformation techniques and thus is inexpensive enough to be incorporated into most commercial compilers. This section provides an overview of the transformation without going into details of the algorithms. For more details, see [28].

We use an example, LU factorization without pivoting, to illustrate the process of applying dependence hoisting. Figure 2 shows two equivalent versions of this linear algebra code (different loop orderings of non-pivoting LU were discussed in more detail by Dongarra, Gustavson and Karp [8]). The KJI version in Figure 2(a) is used in the LINPACK collection [7]. The JKI version in (b) is a less commonly used *deferred-update* version which defers all the updates to each column of the matrix until immediately before the scaling of that column. Specifically, at each iteration of the outermost j loop in (b), 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. This JKI form has a better memory hierarchy performance than the KJI form,

as shown in Section 5.2.

Although the two versions of non-pivoting LU in Figure 2 are equivalent, they have dramatic different loop structures. To translate between these two versions, the $k(s_2)$ and $j(s_2)$ loops must be interchanged, and one of the loops must be fused with the $k(s_1)$ loop at the outermost loop level. Since the $k(s_2)$ and $j(s_2)$ loops cannot be made perfectly nested in either (a) or (b), it is impossible to translate between these two versions using the traditional loop interchange and fusion techniques [24, 25, 20, 4], because these techniques apply only to perfect loop nests.

We show in the following how to apply dependence hoisting, a combined interchange and fuse transformation for arbitrarily nested loops, to facilitate the translation. Specifically, we show how to translate the KJI form in Figure 2(a) to the JKI form in (b). The translation requires distributing the outermost k loop in (a) in order to fuse the distributed $k(s_1)$ loop with the $j(s_2)$ loop and then shift the fused loop to the outermost loop level. Section 3.1 illustrates how to automatically recognize the safety of fusing the $k(s_1)$ and the $j(s_2)$ loops in (a) and then shifting them to the outermost loop level. Section 3.2 describes how to perform the actual interchange and fusion transformations.

Note that although dependence hoisting is highly effective in transforming non-perfectly nested loops, it can also be applied to simpler loop structures such as perfectly nested loops. For example, it can be used to facilitate multi-level fusion of a code segment, as discussed in Section 4.

3.1 Dependence Hoisting Analysis

Building on the safety analysis of loop interchanged and fusion in Section 2.3, this section introduces how to identify opportunities of applying dependence hoisting, a combined transformation of loop interchange and fusion on an arbitrary loop structure.

Each dependence hoisting transformation fuses a group of loops at the outermost position of a code segment C (the fused loop will surround all the statements and other loops in C). We use the name *computation slice* (or *slice*) to denote the group of loops to be fused, and use the name *slicing loop* to denote each loop to be fused. Each slicing loop ℓ surrounds a statement s inside the original code segment C, and a small integer is associated with $\ell(s)$ as the *alignment factor* of $\ell(s)$. Each statement s is called the *slicing statement* of its slicing loop. A

computation slice thus has the following attributes:

- *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).

Each computation slice can be used to guide a dependence hoisting transformation, which fuses all the slicing loops with the correct alignments. In order for the transformation to be legal, a computation slice must satisfy the following three conditions: first, it includes all the statements in C; second, all of its 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 s.t. $Ivar(\ell_x) + \textit{slice-align}(s_x) = Ivar(\ell_y) + \textit{slice-align}(s_y)$.

Figure 2(e) shows the legal computation slices for the KJI version of non-pivoting LU in Figure 2(a). These slices are constructed by first finding all the loops that can be legally shifted to the outermost level and then combining those slicing loops that can be legally fused (for more details, see [28]). The safety of the interchange and fusion transformations is determined by examining the dependence and transitive dependence edges for non-pivoting LU using the EDM representation, as shown in Figure 2(c) and (d). The safety conditions are discussed in Section 2.3.

3.2 Dependence Hoisting Transformation

This section applies dependence hoisting to translate the KJI form of non-pivoting LU in Figure 2(a) into the JKI form in (b). As this translation requires fusing the $k(s_1)$ and $j(s_2)$ loops in (a) at the outermost loop level, the computation slice $slice_j$ in Figure 2(e) is used to guide the transformation.

In order to fuse the $k(s_1)$ with the $j(s_2)$ loops in Figure 2(a), the key requirement is to distribute the outermost $k(s_1, s_2)$ loop in (a). Since this k loop carries a dependence cycle connecting s_1 and s_2, the traditional techniques cannot distribute this loop. Dependence hoisting overcomes this difficulty in three steps, as illustrated in Figure 3.

First, dependence hoisting creates a new dummy loop (with index variable x iterating over the union of the iteration ranges of the $k(s_1)$ and $j(s_2)$ loops) surrounding the original code in Figure 3(a). In the same step, it inserts conditionals in (a) so that statement s_1 is executed only when $x = j$ and s_2 is executed only when $x = k$.

Figure 3(b) shows the result of this transformation step, along with the relationship between iteration numbers at the source and sink of each dependence (before and after the transformation).

Now, because the conditionals 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 Figure 3(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 makes it possible for the second step to distribute the $k(s_1, s_2)$, which no longer carries a dependence cycle. The transformed code after distribution is shown in Figure 3(c). Note that this step requires interchanging the order of s_1 and s_2.

Finally, dependence hoisting removes all the redundant conditionals and loops by substituting the index variable x for the index variables of the $k(s_1)$ and $j(s_2)$ loops. In addition, the upper bound for the $k(s_2)$ loop must be adjusted to $x - 1$, in effect because the $j(s_2)$ loop is exchanged outward before the substitution. The transformed code after this cleanup step is shown in Figure 3(d).

The final transformed code in Figure 3(d) is the same as the JKI form of non-pivoting LU in Figure 2(b) 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.

4 Combined Interchange and Multi-Level Fusion

This section presents algorithms that improve the memory hierarchy performance through a combined loop interchange and multi-level fusion strategy, applying the dependence hoisting transformation introduced in Section 3. Dependence hoisting can be applied to facilitate three loop transformations on arbitrary loop structures: loop interchange, fusion and blocking. In this paper, we focus on interchange and fusion optimizations. For strategies on blocking complex loop structures, see [28].

Given a code segment C to optimize, we first construct all the valid computation slices for C. We then perform data reuse analysis and merge the initial computation slices when beneficial. In particular, for each

loop nest in C, we construct one or more computation slices, all of which are then recombined based on the profitability analysis result of applying interchange and fusion for better locality. Finally, we use the re-arranged set of slices to perform dependence hoisting transformations, which then optimize the original code accordingly.

To present the algorithms in more detail, Section 4.1 elaborates how to model loop fusion in terms of computation slices. Section 4.2 then describes how to systematically apply dependence hoisting to achieve aggressive multi-level loop fusion for better memory hierarchy performance.

4.1 Merging Computation Slices

This section presents algorithms to merge *disjunct computation slices* (slices that contain disjunct sets of statements). Because a dependence hoisting transformation fuses all the loops in a computation slice, we can regard each computation slice as representing a single loop, the *fused loop* of the slice. Given two disjunct computation slices, $slice_1$ and $slice_2$, merging the two slices is therefore equivalent to fusing the two loops represented by $slice_1$ and $slice_2$ respectively.

Figure 7(a) presents two algorithms: one determines whether two computation slices can be legally merged; the other performs the actual merging of the two computation slices.

The function *Comp-Slice-Fusible* determines whether two disjunct computation slices can be legally merged. Because these slices belong to different loop nests at the same loop level, there can be no dependence cycle connecting them. The algorithm 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 arguments can be switched without sacrificing generality). The algorithm examines each dependence edge D from a statement $s_x \in slice_1$ to $s_y \in slice_2$. If the dependence condition from loop ℓ_x (*slice-loop*(s_x) in $slice_1$) to loop ℓ_y (*slice-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 *align* so that

$$align \geq Align(D(\ell_x, \ell_y)) + slice\text{-}align(s_x) - slice\text{-}align(s_y). \tag{5}$$

If the algorithm succeeds in finding a valid fusion alignment *align* after examining all the dependence edges, the

two computation slices should be merged so that

$$Ivar(\ell_{f1}) = Ivar(\ell_{f2}) + align, \tag{6}$$

where ℓ_{f1} and ℓ_{f2} represent the fused loops of $slice_1$ and $slice_2$ respectively. Equation (6) indicates that the computation slice $slice_2$ needs to be aligned by the factor $align$ before being fused with $slice_1$.

To prove that the algorithm *Comp-Slice-Fusible* is correct, we show that after merging $slice_1$ and $slice_2$ according to Equation (6), no dependence edge from $slice_1$ to $slice_2$ is reversed. 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 computation slices in Section 3.1:

$$Ivar(\ell_{f1}) = Ivar(\ell_x) + slice\text{-}align(s_x) \tag{7}$$

$$Ivar(\ell_{f2}) = Ivar(\ell_y) + slice\text{-}align(s_y) \tag{8}$$

After substituting the above two equations for ℓ_{f1} and ℓ_{f2} in Equation (6), the following relation between the slicing loops ℓ_x and ℓ_y is satisfied:

$$Ivar(\ell_x) + slice\text{-}align(s_x) = Ivar(\ell_y) + slice\text{-}align(s_y) + align \tag{9}$$

Now consider each dependence EDM D from s_x to s_y. Because the fusion alignment $align$ satisfies Equation (5), substituting this inequality for $align$ in Equation (9) obtains

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

which is equivalent to

$$Ivar(\ell_x) \leq Ivar(\ell_y) + Align(D(\ell_x, \ell_y)) \tag{10}$$

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 7(a) performs the actual merging 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 fusion alignment *align* so that the fusion relation specified by Equation (5) is satisfied.

4.2 Optimization Algorithms

This section presents the algorithm that achieves a combined interchange and multi-level fusion optimization to enhance locality. Figure 7(b) shows the optimization algorithm, where the function *Optimize-Code-Segment* is the main function. The algorithm optimizes a code segment C in the following steps.

Step (1) As shown in Figure 7(b), the algorithm first distributes the input code segment C into strongly connected components. For each distributed loop nest, it then finds all the computation slices that can be shifted to the outermost level of C. The construction of these slices is described in Section 3.1. The distributed loop nests will later be re-fused when the collected computation slices are merged.

As the result of invoking function *Hoisting-Analysis(L, slice-nest)*, one or more computation slices is constructed for the loop nest L and is placed into a list *slice-nest*. These slices contain the same set of statements, and together, they can be seen as forming a loop nest. We thus name this list of slices a *slice nest*. For each constructed slice nest, we then determine its best nesting order based on data reuse information. By counting the number of data reuses carried by the slicing loops of each slice, we arrange the order of slices in *slice-nest* so that the slices that carry more reuses will be nested inside.

Step (2) After Step (1), a collection of slice nests have been constructed for the input code segment C. These slice nests are disjunct in that each nest contains a disjunct set of statements. Because there are no dependence cycles connecting these slice nests, they can be further merged to achieve multi-level loop fusion optimization on the original code.

In order to merge the disjunct slice nests constructed in Step (1), Step (2) of the algorithm constructs a *slice-fusion dependence graph* to model the dependences between each pair of the slice nests. This graph is then used to fuse the slice nests in Step (3), which adapts the typed-fusion algorithm by Kennedy and McKinley [15] to apply to the new fusion dependence graph.

The input of the original typed-fusion algorithm [15] is a loop-fusion dependence graph, where each vertex

of the graph represents a loop, and an edge is put from vertex x to y in the graph if there are dependences from statements inside loop x to statements inside loop y. The edge is annotated as a *bad edge* if the dependences prevent the two loops (x and y) from being legally fused. The fusion algorithm then clusters the vertices that are not connected by fusion-preventing *bad* paths.

To adapt the fusion dependence graph for merging nests of computation slices, we modify the graph so that each vertex is a slice nest (a set of computation slices containing the same statements) instead of a single loop. An edge is put from vertex x to y if there are dependences from statements in the slice nest x to statements in the slice nest y, and the edge is annotated as a *bad edge* if these two nests cannot be legally merged. The safety of merging two slice nests is determined by applying the function *Comp-Slice-Fusible* in Figure 7(a) to each pair of computation slices from the two nests. The edge between two slice nests x and y is annotated as a *bad edge* if no slices from the two nests can be fused.

Step (3.1) After constructing the slice-fusion dependence graph in Step (2), this step applies the typed-fusion algorithm by McKinley and Kennedy [15]) to cluster vertices (each vertex representing a slice nest) in the slice-fusion dependence graph. The typed-fusion algorithm is a linear algorithm that takes as input a fusion dependence graph. It then aggressively clusters vertices in the graph so that there are no cycles between clusters and there are no bad edges inside each cluster.

For each set of vertices clustered by the typed-fusion algorithm, Step (3.1) tries to merge as many slice nests as possible by considering each pair of vertices that can be collapsed into a single vertex without creating cycles in the cluster. It then tries to fuse the slice nests, $slicenest_1$ and $slicenest_2$, in these two vertices by invoking the function *Fuse-Slice-Nest*, which fuses the nests only when profitable. If the fusion succeeds, the function *Fuse-Slice-Nest* also collapses the corresponding vertices into a single vertex in the fusion dependence graph.

Step (3.2) This step is essentially the same as Step (3.1) except that now the input slice-fusion dependence graph G is reversed (the direction of each edge in G is reversed). This step is necessary because some of the clustered vertices in Step (3.1) may not be successfully collapsed if the function *Fuse-Slice-Nest* determines that it is not profitable to perform the fusion. These left-out vertices, however, may be fused with vertices in different clusters. Step (3.2) ensures that these opportunities are exploited by re-clustering the fusion dependence graph

G. The reversal of G ensures a different clustering than that in Step (3.1) [2].

Function *Fuse-Slice-Nest* This function is invoked by Steps (3.1) and (3.2) of the main function *Optimize-Code-Segment* to fuse slice nests and to collapse vertices in the slice-fusion dependence graph.

Given the two slice nests, $slicenest_1$ and $slicenest_2$, function *Fuse-Slice-Nest* first examines each pair of computation slices, $slice_1 \in slicenest_1$ and $slice_2 \in slicenest_2$, to determine whether $slice1$ should be fused with $slice_2$ and to determine the fusion alignment if necessary. Although $slicenest_1$ and $slicenest_2$ are not connected by *bad* paths, all slices in them may not be legally fused. The function therefore first invokes function *Comp-Slice-Fusible*, which is defined in Figure 7(a), to determine whether $slice_1$ and $slice_2$ can be legally merged. If the answer is "yes", *Fuse-Slice-Nest* then checks profitability of merging the two slices. If the fusion of these two slices is in conflict with their nesting order arranged by Step (1) of the main function *Optimize-Code-Segment*, the data reuse information for $slice_1$ and $silce_2$ is computed, and the fusion is performed only if the data reuse gained from fusion outweighs the reuse lost from loop interchange. The actual fusion of $slice_1$ and $slice_2$ is performed by invoking the function *Fuse-Comp-Slice* defined in Figure 7(a).

Note that the function *Fuse-Slice-Nest* can partially fuse the two slice nests, $slicenest_1$ and $slicenest_2$, and the left-over slices (slices that have not participated in any fusion) will be placed inside other slices that contain more statements. This forced nesting order is guaranteed to be beneficial, because the performance tradeoff between fusion and interchange has already been evaluated before each pair of slices are merged.

Step (4) This step of the algorithm uses the fused slice nests to transform the original code C, realizing the combined loop interchange and multi-level fusion optimization.

The algorithm uses a variable C_1 to keep track of the code segment to transform by each slice nest. It traverses the computation slices in each nest in the reverse of their nesting order. After using each computation slice $slice_i$ to guide a dependence hoisting transformation, all the slicing loops in $slice_i$ have been fused into a single loop ℓ_f, which now surrounds the original code segment C_1. The algorithm then sets C_1 to be ℓ_f and then uses C_1 as input for further dependence hoisting transformations, which will shift other loops outside ℓ_f. As the

[2]The reversal of the slice-fusion dependence graph will not cause the slice nests to be clustered incorrectly. For more detail, see the typed-fusion algorithm in [15].

result, all the slicing loops in each computation slice are fused into a single loop, and the fused loops are nested in the desired order.

5 Experimental Results

This section presents experimental results that verify the effectiveness of our combined interchange and multi-level fusion framework, as described in Section 3 and 4.

To illustrate the effectiveness of *dependence hoisting* in transforming arbitrary loop structures, we show the results of applying this technique to optimize four linear algebra kernels, Cholesky, QR, LU factorization without pivoting, and LU factorization with partial pivoting. These kernels are important linear algebra subroutines that are widely used in scientific computing applications. Furthermore, they all contain complex loop structures that are generally considered difficult to optimize automatically. Applying the dependence hoisting technique, we are able to automatically optimize all these kernals and achieve significant performance improvements. Since loop blocking is not the focus of this paper, here we present the performance results of applying only loop interchange to these kernals. The result of applying loop blocking are presented in detail in our earlier work [28].

To illustrate the advantage of combining interchange with multi-level fusion, we present the performance results of four application benchmarks, *tomcatv*, *Erlebacher*, *SP*, and *twostages*, all of which benefit significantly from loop fusion. More information for these applications is provided in Table 1. For each benchmark, we compare the performance achieved from applying combined interchange and fusion with the performance of the original version, and with the performance achieved from applying interchange and fusion separately. For two of these benchmarks, combined interchange and fusion yields better performance than applying interchange and fusion separately, indicating that it is beneficial to evaluate the compound effect of interchange and fusion together.

In the following, Section 5.1 first introduces our compiler implementation and experimental design. Sections 5.2 and 5.3 then present the performance results of the linear algebra kernels and the application benchmarks respectively.

5.1 Experimental Design

We have implemented the dependence hoisting transformation and the combined interchange and fusion framework in a Fortran source-to-source translator. Given an input application, the translator globally optimizes each subroutine by selectively applying three loop transformations: loop interchange, blocking and fusion. Loop interchange is applied to shift loops that carry more data reuses inside, loop blocking is applied to exploit data reuses carried by outer loops, and loop fusion is applied to exploit data reuses across different loop nests. All three transformations are implemented in terms of computation slices and are carried out by dependence hoisting transformations. To evaluate only the impact of loop interchange and fusion, we have turned off blocking in the translator. The strategies for applying combined interchange and multi-level fusion is described in Section 4.

To compare different optimization strategies, we have implemented in the translator multiple heuristics of applying fusion and interchange. We present the performance of two strategies: the first strategy evaluates the compound effect of loop interchange and fusion collectively; the second strategy first applies loop interchange, then fuses loop nests only when the fusion does not change the pre-arranged loop nesting order. The second strategy is equivalent to the previous interchange and fusion algorithms [6, 15, 14, 10, 24, 20], which separate loop fusion and interchange. By comparing these two strategies, we therefore compare our combined interchange and fusion technique with the previous separate interchange and fusion techniques.

We transcribed the original versions of the four linear algebra kernels (*Cholesky*, *QR*, *LU factorization without pivoting*, and *LU factorization with partial pivoting*) from the simple versions found in Golub and Van Loan [11]. The original version of LU without pivoting is shown in Figure 2(a). The original versions of other kernels are written similarly. The code for QR is based on Householder transformations [11].

We downloaded the original versions of the application benchmarks in Table 1 from various benchmark suites. The benchmark *tomcatv* is obtained from SPEC95. The benchmarks *Erlebacher* and *SP* are obtained from ICASE and NAS benchmark suites respectively. The benchmark *twostages* is a subroutine from a large weather prediction system. It is a portion of the Runge-Kutta advection scheme from Wilhelmson's COMMAS meteorology code, and is obtained from NCSA (the National Center for Supercomputing Applications). When given as input to the translators, all these benchmarks are used in their original forms with essentially no modification.

We measured the performance results of all the benchmarks 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.

We compiled all the benchmarks (including their original versions and automatically optimized versions) 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 optimizations such as global fusion/distribution or loop interchange. All the versions were optimized at the same level by the SGI compiler, guaranteeing a fair comparison. Each measurement is repeated 5 or more times and the average across these runs is presented. The variations across runs are very small (within 1%).

5.2 Performance of Linear Algebra Kernels

Figure 5 shows the performance results of the linear algebra kernels using a 1000*1000 matrix. Each benchmark has two versions: the original version transcribed from Golub and Van Loan [11], and the optimized version after applying loop interchange by our translator, which counts the number of data reuses carried by each computation slice and then arranges the nesting order of these slices in increasing order of the carried reuses. We denote these two versions as the original and interchanged versions respectively. Each set of measurements is normalized to the performance of the original version.

All the interchanged versions are able to perform better than the original versions except for non-pivoting LU, for which the two interchanged loops carry similar data reuses, and consequently the interchanged version does not manifest better locality unless the matrix size is $\geq 2000^2$. The performance improvements for *Cholesky*, *QR* and *pivoting LU* are shown uniformly in the cycle count graphs. As the translator only interchanged the outer loops in these kernels (the original innermost loops stay innermost in all cases). The better locality is shown in the L2 cache-miss graphs for *Cholesky* and *QR*, and is shown in the TLB miss graph for *LU with partial pivoting*. If blocking optimization is further applied, the performance improvements can rise up to 5-10 factors, see [28].

Because blocking is the most effective optimization for these linear algebra kernels, the performance comparison between the interchanged versions and original versions, as shown in Figure 5, is not significant. However, loop interchange is the base transformation for blocking. Without the ability to apply interchange to these kernels, blocking would be impossible. The advantage of dependence hoisting lies in that it can foresee all the equivalent nesting orders of an arbitrary loop structure, even when the loop nest is non-perfectly nested. As people tend to design algorithms in the most intuitive versions, for which an alternative version may not be obvious, it should be left to the optimizing compilers to translate these versions into equivalent ones with better performance. The novel transformation introduced in this paper contributes to this objective by offering all the equivalent versions of an arbitrarily nested loop structure. A compiler can then choose from these equivalent versions the one with the best performance.

5.3 Performance of Application Benchmarks

Figure 6 shows the performance results of applying both loop interchange and fusion optimizations to four application benchmarks, *tomcatv*, *Erlebacher*, *SP*, and *twostages*, which are described in Table 1 and in Section 5.1. For each benchmark, we present the performance of four versions: the original version, the version optimized by loop interchange only, the version optimized by first applying loop interchange and then applying fusion to loops at the same level, and the version optimized by applying combined loop interchange and fusion as described in Section 4. In the following discussion, we denote these versions as the original version, interchanged version, same-level fused version, and multi-level fused version respectively.

From Figure 6, for three of the four benchmarks, loop interchange has rearranged the original nesting order and has yielded better performance. For *tomcatv*, no loop interchange is performed because the original version already has the best nesting order. By applying loop interchange before fusion, we ensure that the best order is arranged for each loop nest. The fusion algorithms thus are guaranteed to have the right loop nesting order to start with.

In Figure 6, the multi-level fused versions of two applications, *Erlebacher* and *SP*, have achieved better performance than the same-level fused versions. Here the combined interchange and fusion strategy is able to exploit extra data reuses because for a few loop nests in these two applications, multiple nesting orders can yield

similar performance. The interchanged versions of these two applications have selected the nesting orders based on the local data reuse analysis inside each nest, and the chosen nesting orders happen to prevent these nests from being further fused when same-level fusion is applied. Because multi-level fusion strategy can evaluate the compound effect of fusion and interchange together, it can realize in these cases that fusion is more favorable than maintaining the nesting orders arranged by local loop interchange analysis. It therefore re-arranged the original nesting orders to facilitate fusion.

For the other benchmarks, *tomcatv* and *twostages*, both the same-level fusion and multi-level fusion strategies have produced the same optimized code and therefore have yielded the same performance. For these two benchmarks, although certain loops can be further fused after same-level fusion, the benefit of extra fusion does not outweigh the loss of changing the original nesting order. The multi-level fusion algorithm thus decided not to perform these fusions.

Note that besides exploiting data reuses across multiple loop nests, loop fusion can also facilitate other optimizations such as array contraction [10]. It is straight forward to incorporate these optimizations into our framework. Because the combined interchange and multi-level fusion framework is not sensitive to the original nesting order of individual loop nests, it can incorporate both local and global transformations. By evaluating the compound effect of these transformations, the framework ensures a global optimization by putting aside sub-optimal transformations.

6 Related Work

A set of unimodular and single loop transformations, such as loop blocking, fusion, distribution, interchange, skewing and index-set splitting [24, 17, 20, 5, 9, 25, 4], have been proposed and widely used to improve both the memory hierarchy and parallel performance of applications. This paper focuses on improving two of these transformations: loop interchange and fusion.

Previous loop interchange transformation techniques [24, 20] can be applied only to perfectly nested loops, and the safety of interchange is determined by computing the dependence relations between iterations of common loops surrounding statements. Although loop distribution and fusion can be applied to transform simple loop

nests into perfectly nested, on more complicated loop structures, such as the one for non-pivoting LU in Figure 2, previous transformation techniques often fail even though a solution exists. This paper has presented a new loop transformation, dependence hoisting, which facilitates the interchange and fusion of arbitrarily nested loops. Our technique is independent of the original loop structures of programs and can enable transformations that are not possible through previous unimodular and single loop transformation techniques.

Loop fusion can be applied both to improve memory hierarchy performance in sequential programs and to reduce synchronization overhead in parallel programs. Optimal fusion for either locality or parallelism is NP-complete in general [6]. Kennedy and McKinley [15] proposed a *typed fusion* heuristic, which can achieve maximal fusion for loops of a particular type in linear time. Kennedy [14] also proposed a weighted fusion algorithm that aggressively fuses loops connected by the most heavy edges (edges that signal the most data reuses between loops). Gao, Olsen, Sarkar and Thekkath [10] proposed a heuristic that applies loop fusion to increase the opportunities of applying array contraction and in turn to reduce the number of array accesses. Manjikian and Abdelrahman [19] introduced a *shift-and-peel* transformation, which aligns iterations of loops before fusion. Similar to Manjikian and Abdelrahman, we also incorporate loop fusion with loop shifting. In addition, we combine loop fusion with loop interchange, and collectively evaluate the performance impact of both transformations. As the result, we have a framework that simultaneously fuses multi-levels of loops through a combined interchange and fusion strategy.

Several general loop transformation frameworks [16, 21, 1, 18, 23] are theoretically more powerful but are also more expensive than the techniques introduced in this paper. In particular, Pugh [21] proposed a framework that finds a schedule for each statement describing the moment each statement instance will be executed. Kodukula, Ahmed and Pingali [16] 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, where a mapping from the iteration spaces of statements into the *data* spaces is computed. Lim, Cheong and Lam [18] 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. All these transformation frameworks compute a mapping from the iteration spaces of statements

into some other space. The computation of these mappings is expensive and generally requires special integer programming tools such as the Omega library [12]. In contrast, this paper seeks simpler solutions with a much lower compile-time overhead.

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 [22, 13]. 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 uses a much simpler dependence representation and a much more efficient transitive dependence analysis algorithm. Our techniques make transitive dependence analysis fast enough for incorporation in production compilers.

7 Conclusion

We present a novel loop transformation, dependence hoisting, that achieves a combined interchange and fusion transformation on arbitrarily nested loops. Based on this transformation, we present a combined interchange and multi-level fusion optimization framework for applications with arbitrary loop structures. This framework collectively evaluates the benefit of loop interchange and fusion and by doing so, can ensure better performance than that was possible by previous techniques, which consider each transformation separately.

We have implemented our transformation techniques and have applied them to optimize a collection of benchmarks. Using dependence hoisting, we are able to optimize four linear algebra kernels, Cholesky, QR, LU without pivoting and LU with partial pivoting, which contain complex loop structures that were generally considered difficult to optimize automatically. In addition, by combining interchange and multi-level fusion, we are able to secure a higher performance level than that by applying these two transformations separately. These results indicate that our techniques are highly effective, both in transforming complex loop structures in small

kernels and in securing a globally optimal performance for general application benchmarks.

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

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

[4] 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.

[5] 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.

[6] A. Darte. On the complexity of loop fusion. *Parallel Computing*, 26(9):1175–1193, 2000.

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

[8] 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.

[9] 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.

[10] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *The 5th Workshop on Languages and Compiler for Parallelism*, Springer-Verlag, 1992.

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

[12] 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.

[13] 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.

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

[15] 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).

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

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

[24] 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.

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

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

[27] 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.

[28] Q. Yi and K. Kennedy. Transforming complex loop nests for locality. Technical Report TR02-386, Computer Science Dept., Rice University, Feb. 2002.

```
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
do 50 j=1,N
do 50 i=1,N
   duz(i,j,N) = (duz(i,j,N) - tot(i,j))*b(N)
50 continue
do 60 j=1,N
do 60 i=1,N
   duz(i,j,N-1)=duz(i,j,N-1) - e(N-1)*duz(i,j,N)
60 continue
do 70 k=N-2, 1, -1
do 70 j=1,N
do 70 i=1,N
   duz(i,j,k) = duz(i,j,k) - c(k)*duz(i,j,k+1)
                - e(k)*duz(i,j,N)
70 continue
```

(a) original code

```
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
do 50 j=1,N
do 50 i=1,N
   duz(i,j,N) = (duz(i,j,N) - tot(i,j))*b(N)
50 continue
do 60 j=1,N
do 60 i=1,N
   duz(i,j,N-1)=duz(i,j,N-1) - e(N-1)*duz(i,j,N)
60 continue
do 70 j=1,N
do 70 k=N-2, 1, -1
do 70 i=1,N
   duz(i,j,k) = duz(i,j,k) - c(k)*duz(i,j,k+1)
                - e(k)*duz(i,j,N)
70 continue
```

(b) loop interchange

```
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
do 70 j=1,N
  do 60 i=1,N
     duz(i,j,N) = (duz(i,j,N) - tot(i,j))*b(N)
     duz(i,j,N-1)=duz(i,j,N-1) - e(N-1)*duz(i,j,N)
  60 continue
  do 70 k=N-2, 1, -1
  do 70 i=1,N
     duz(i,j,k) = duz(i,j,k) - c(k)*duz(i,j,k+1)
                  - e(k)*duz(i,j,N)
70   continue
```

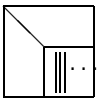
(c) interchange + traditional fusion

```
do 70 j=1,N
  do 40 k=1,N-1
  do 40 i=1,N
     tot(i,j) = tot(i,j) + d(k)*duz(i,j,k)
  40 continue
  do 60 i=1,N
     duz(i,j,N) = (duz(i,j,N) - tot(i,j))*b(N)
     duz(i,j,N-1)=duz(i,j,N-1) - e(N-1)*duz(i,j,N)
  60 continue
  do 70 k=N-2, 1, -1
  do 70 i=1,N
     duz(i,j,k) = duz(i,j,k) - c(k)*duz(i,j,k+1)
                  - e(k)*duz(i,j,N)
70   continue
```

(d) multi-level fusion

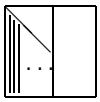
Figure 1: Example to illustrate the advantage of multi-level fusion over traditional single-level fusion

```
 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 version

```
 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
```



(b) JKI version

$$\mathbf{d}(\mathbf{s_1},\mathbf{s_2}) = \left\{ \begin{array}{c} \quad k(s_2)\ j(s_2)\ i(s_2) \\ k(s_1) \\ i(s_1) \end{array} \begin{pmatrix} \quad k(s_2)\ j(s_2)\ i(s_2) \\ =0 \le -1 \le -1 \\ \ge 1 \quad * \quad =0 \end{pmatrix} \right\}$$

$$\mathbf{d}(\mathbf{s_2},\mathbf{s_1}) = \left\{ \begin{array}{c} \quad k(s_1)\ i(s_1) \\ k(s_2) \\ j(s_2) \\ i(s_2) \end{array} \begin{pmatrix} \quad k(s_1)\ i(s_1) \\ \le -1 \le -2 \\ =0 \ \le -1 \\ \ge 1 \ \ =0 \end{pmatrix} \begin{pmatrix} \quad k(s_1)\ i(s_1) \\ \le -1 \le -2 \\ =0 \ \le -1 \\ =0 \ \le -1 \end{pmatrix} \right\}$$

$$\mathbf{d}(s_2,s_2) = \left\{ \begin{array}{c} \quad k(s_2)\ j(s_2)\ \ i(s_2) \\ k'(s_2) \\ j'(s_2) \\ i'(s_2) \\ k'(s_2) \\ j'(s_2) \\ i'(s_2) \end{array} \begin{pmatrix} \quad k(s_2)\ j(s_2)\ \ i(s_2) \\ \le -1 \le -2 \le -2 \\ \ge 1 \ \ =0 \ \ * \\ \ge 1 \ \ * \ \ =0 \end{pmatrix} \begin{pmatrix} \quad k(s_2)\ j(s_2)\ \ i(s_2) \\ \le -2 \le -3 \le -3 \\ =0 \ \le -1 \le -1 \\ \ge 1 \ \ * \ \ =0 \end{pmatrix} \begin{pmatrix} \quad k(s_2)\ j(s_2)\ \ i(s_2) \\ \le -2 \le -3 \le -3 \\ \ge 1 \ \ =0 \ \ * \\ =0 \ \le -1 \le -1 \end{pmatrix} \right\}$$

(c) dependence edges

$$\mathbf{td}(\mathbf{s_1},\mathbf{s_1}) = \left\{ \begin{array}{c} \quad k(s_1)\ i(s_1) \\ k'(s_1) \\ i'(s_1) \end{array} \begin{pmatrix} \quad k(s_1)\ i(s_1) \\ \le -1 \le -2 \\ \ge 1 \ \ =0 \end{pmatrix} \begin{pmatrix} \quad k(s_1)\ i(s_1) \\ \le -1 \le -2 \\ \le 0 \ \ \le -1 \end{pmatrix} \right\}$$

$$\mathbf{t}d(s_1,s_2) = \left\{ \begin{array}{c} \quad k(s_2)\ j(s_2)\ i(s_2) \\ k(s_1) \\ i(s_1) \end{array} \begin{pmatrix} \quad k(s_2)\ j(s_2)\ i(s_2) \\ \le 0 \le -1 \le -1 \\ \ge 1 \ \ * \ \ =0 \end{pmatrix} \begin{pmatrix} \quad k(s_2)\ j(s_2)\ i(s_2) \\ \le -1 \le -2 \le -2 \\ \le 0 \ \ \le -1 \le -1 \end{pmatrix} \right\}$$

$$\mathbf{td}(\mathbf{s_2},\mathbf{s_1}) = \left\{ \begin{array}{c} \quad k(s_1)\ i(s_1) \\ k(s_2) \\ j(s_2) \\ i(s_2) \end{array} \begin{pmatrix} \quad k(s_1)\ i(s_1) \\ \le -1 \le -2 \\ \le 0 \ \ \le -1 \\ \ge 1 \ \ =0 \end{pmatrix} \begin{pmatrix} \quad k(s_1)\ i(s_1) \\ \le -1 \le -2 \\ \le 0 \ \ \le -1 \\ \le 0 \ \ \le -1 \end{pmatrix} \right\}$$

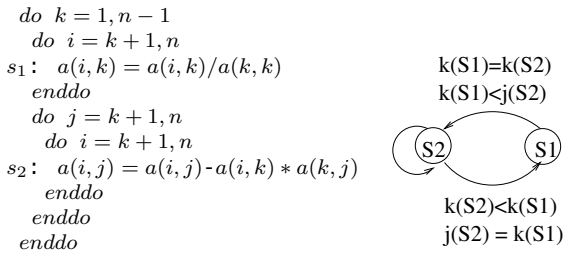
$$\mathbf{td}(\mathbf{s_2},\mathbf{s_2}) = \left\{ \begin{array}{c} \quad k(s_2)\ j(s_2)\ \ i(s_2) \\ k'(s_2) \\ j'(s_2) \\ i'(s_2) \\ k'(s_2) \\ j'(s_2) \\ i'(s_2) \end{array} \begin{pmatrix} \quad k(s_2)\ j(s_2)\ \ i(s_2) \\ \le -1 \le -2 \le -2 \\ \ge 1 \ \ =0 \ \ * \\ \ge 1 \ \ * \ \ =0 \end{pmatrix} \begin{pmatrix} \quad k(s_2)\ j(s_2)\ \ i(s_2) \\ \le -2 \le -3 \le -3 \\ \le 0 \ \ \le -1 \le -1 \\ \ge 1 \ \ * \ \ =0 \end{pmatrix} \begin{pmatrix} \quad k(s_2)\ j(s_2)\ \ i(s_2) \\ \le -2 \le -3 \le -3 \\ \le 0 \ \ \le -1 \le -1 \\ \le 0 \ \ \le -1 \le -1 \end{pmatrix} \right\}$$

(d) transitive dependence edges

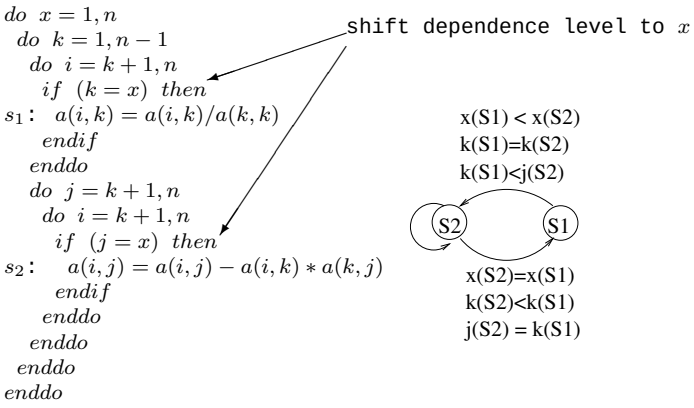
$slice_j$:	$slice_k$:	$slice_i$:
$stmt\text{-}set = \{s_1, s_2\}$	$stmt\text{-}set = \{s_1, s_2\}$	$stmt\text{-}set = \{s_1, s_2\}$
$slice\text{-}loop(s_1) = k(s_1)$; $slice\text{-}align(s_1) = 0$	$slice\text{-}loop(s_1) = k(s_1)$; $slice\text{-}align(s_1) = 0$	$slice\text{-}loop(s_1) = i(s_1)$; $slice\text{-}align(s_1)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

(e) computation slices

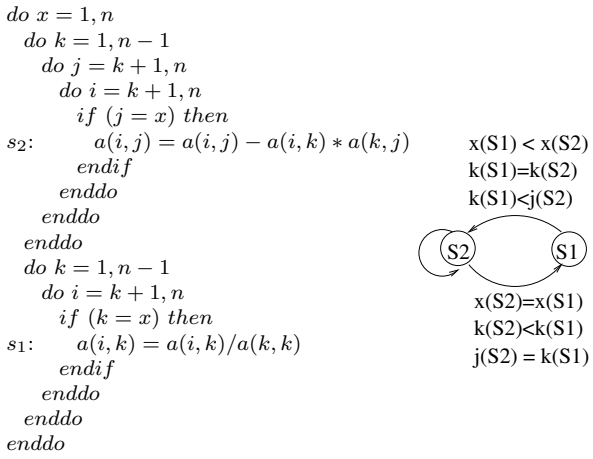
Figure 2: Dependence hoisting analysis for non-pivoting LU



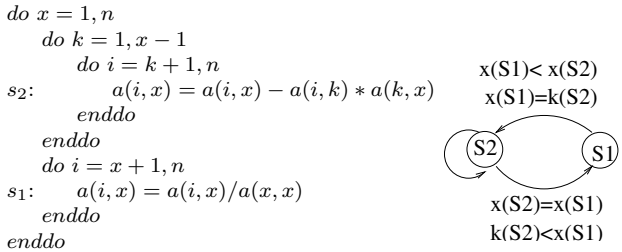
(a) original code



(b) after shifting dependence levels



(c) after distributing loops



(d) after cleanup

Figure 3: Dependence hoisting transformation for non-pivoting LU

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 $slice$

(a) Merging computation slices

Optimize-Code-Segment(C)
(1) *slicenest-vec* = ∅; Distribute loop nests in C
for (each distributed loop nest L) do
Hoisting-Analysis(L**,** *slicenest***)**
Arrange the best nesting order for *slicenest*
Add *slicenest* to *slicenest-vec*
(2) Construct fusion dependence graph G for *slicenest-vec*
(3.1) Apply typed-fusion algorithm to cluster nodes in G
For (each clustered group *slicenest-group*) do
For (each pair of nodes $slicenest_1$ and $slicenest_2$
if (fusing $slicenest_1$ and $slicenest_2$ will not create cycles)
Fuse-Slice-Nodes(G, $slicenest_1$, $slicenest_2$)
(3.2) G' = reverse edges in G
Apply typed-fusion algorithm to re-cluster nodes in G'
For (each clustered group *slicenest-group*) do
For (each pair of nodes $slicenest_1$ and $slicenest_2$
if (fusing $slicenest_1$ and $slicenest_2$ will not create cycles)
Fuse-Slice-Nodes(G, $slicenest_1$, $slicenest_2$)
(4) For (each $slicenest \in G$) do
C_1 = code segment to transform by *slicenest*
For (each $slice_i \in slicenest$ in reverse nesting order)
Hoisting-Transformation(C_1**,** $slice_i$**)**
C_1 = the new fused loop ℓ_f of $slice_i$

Fuse-Slice-Nodes(G**,** $slicenest_1$**,** $slicenest_2$**)**
(1) $fusednest = \emptyset$
For (each $slice_1 \in slicenest_1$)
For (each $slice_2 \in slicenest_2$)
If (!**Comp-Slice-Fusible(**$slice_1$**,** $slice_2$**,**$align$**)**) continue
If ($slice_1$ and $slice_2$ are at different nesting level)
Compute data reuses between $slice_1$ and $slice_2$
If (reuses gained from fusion < reuses lost from interchange)
continue
$slice$ =**Fuse-Comp-Slice(**$slice_1$**,** $slice_2$**,**$align$**)**
Add $slice$ into $fusednest$
If ($fusednest == \emptyset$) then return
(2) Add a new node $N = fusednest$ to G
For (each edge e incident to $slicenest_1$ or $slicenest_2$) do
change e to be incident to N instead

(b) multi-Level fusion

Figure 4: Combined loop interchange and multi-Level fusion framework

- Versions: o—original; s—after loop interchange;
- Benchmarks (matrix size 1000*1000): Cholesky—chl; Non-pivoting LU—lu; Pivoting LU—lup; QR—qr;

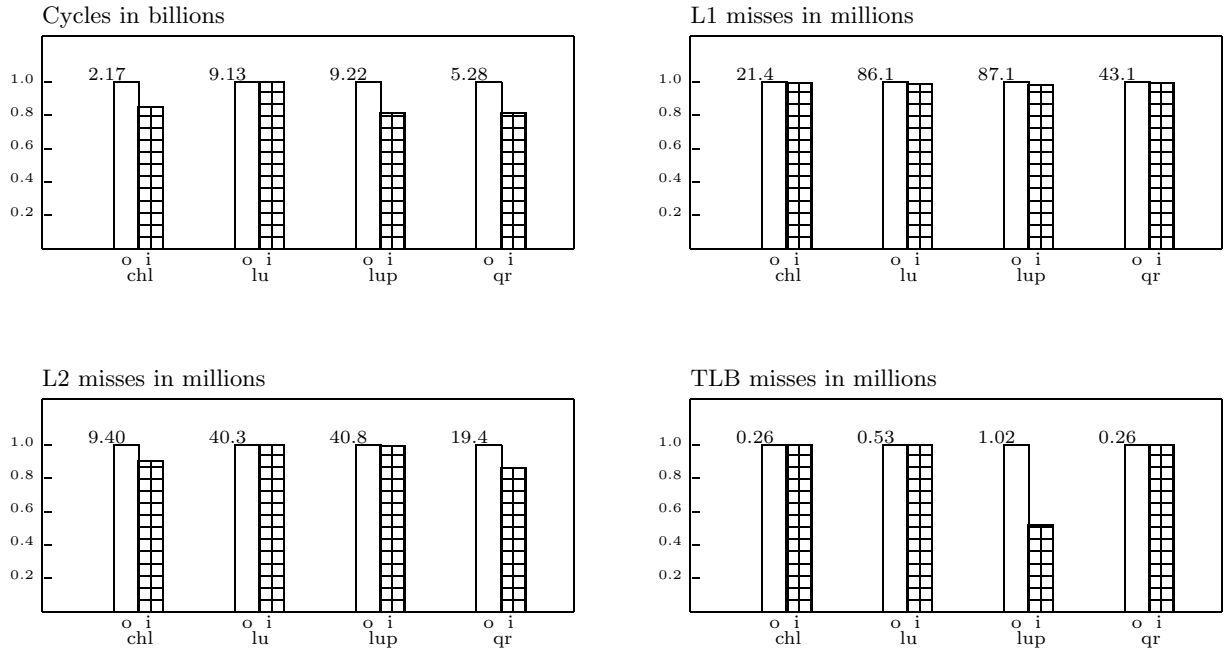


Figure 5: Results from applying interchange to linear algebra kernels

- Versions: o—original; i—optimized with loop interchange; f1—optimized with interchanged and then same-level fusion; f2—optimized with combined interchange and multi-level fusion.
- Benchmarks:

tomcatv:	tcat—513^2 matrix, 750 iterations;
Erlebacher:	Erle—256^3 grid;
SP:	SP—102^3 grid (class B), 3 iterations;
twostages:	2stages—150^3 grid.

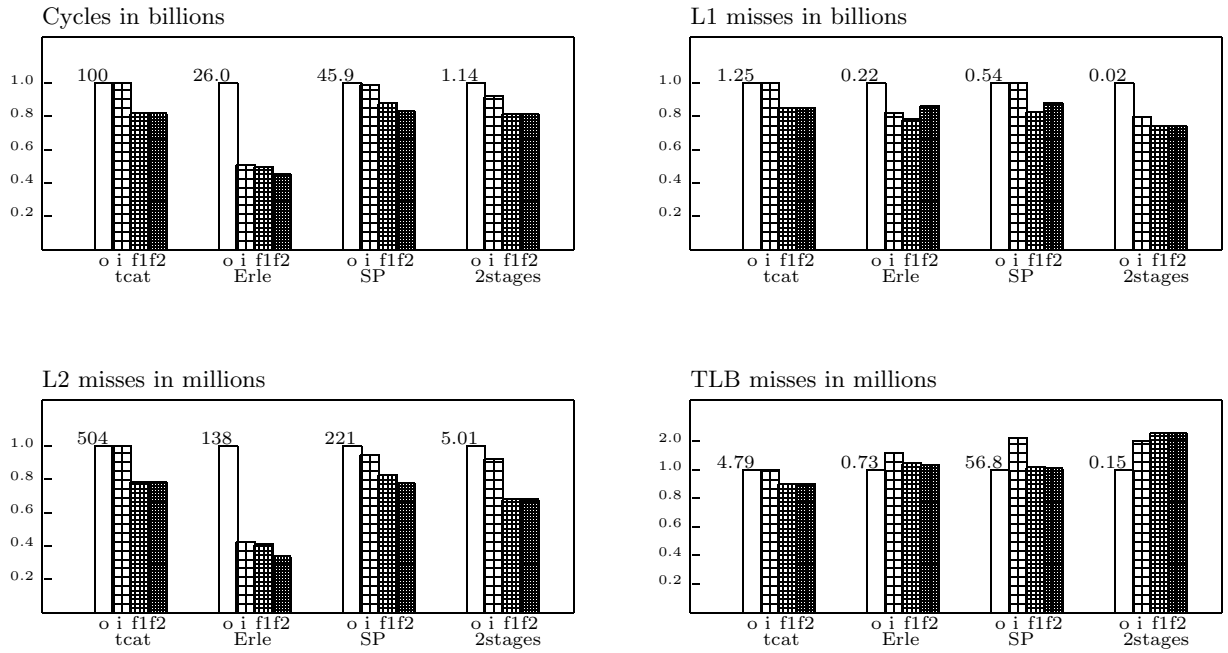


Figure 6: Results from applying combined interchange and fusion to general application benchmarks

Suite	Benchmark	Description	subroutine	No.lines
Spec95	tomcatv	Mesh generation with Thompson solver	all	190
-	twostages	A two stage, three dimensional multi-grid solver from a meteorology code (a weather prediction system)	all	484
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 1: Application benchmarks used in evaluation