Improving Performance with Integrated Program Transformations

Apan Qasem Guohua Jin John Mellor-Crummey
Department of Computer Science
Rice University
Houston, Texas
{qasem,jin,johnmc}@cs.rice.edu

July 3, 2003

Abstract

Achieving a high fraction of peak performance on today's computer systems is difficult for complex scientific applications. To do so, an application's characteristics must be tailored to exploit the characteristics of its target architecture. Today, commercial compilers do not adequately tailor programs automatically; thus, application scientists must settle for lackluster performance or manually transform their codes into a form that is complex and unmaintainable. In this paper, we describe a prototype source-to-source transformation tool that enables application scientists to achieve high performance for scientific codes without changing their natural coding style. Our tool supports a rich, integrated collection of optimizing transformations and provides users with precise control over how these optimizations should be applied. In preliminary experiments with the Runga-Kutta advection core from the NCOMMAS code for mesoscale weather modeling and Livermore Loop 18, we have used our tool to double single-processor performance.

1 Introduction

Every few months, a new generation of microprocessors emerges that is rated to deliver considerably more peak performance than the last. However, obtaining a high fraction of that performance is becoming increasingly difficult. When the characteristics of a scientific application aren't well matched to the characteristics of the architecture on which it executes, performance can fall below 10% of peak. Tailoring a code to achieve the best possible performance on a particular microprocessor-based architecture requires a complex set of program transformations, each designed to optimize for one or more aspects (e.g. registers, cache, TLB, and floating-point pipeline) of the target system. For data-intensive scientific applications, exploiting spatial and temporal reuse of data whereever possible is especially important to minimize the demands that the applications place on the underpowered memory subsystems found in today's microprocessor-based architectures.

Sophisticated scientific codes can be difficult to optimize effectively for microprocessor-based architectures. Often, application developers factor complex calculations into a series of small steps, with each step performing a simple computation to advance some aspect of the program state. When codes are structured in this way, individual loop nests typically consume values of one or more arrays and compute a result into a new array that will be used by subsequent loop nests. Many developers prefer this coding style because it makes applications simple to understand and maintain. However, such codes have a larger memory footprint because of the temporary arrays they use. When data sets are large, they miss opportunities for temporal reuse because values brought in to cache in one loop nest are evicted from cache before they can be reused in another.

While the causes of performance problems, analysis techniques to identify them and optimization techniques to ameliorate them are understood (at least in principle) in the compiler community, automatic optimization of complex scientific codes most often falls significantly short of the best possible performance. A recent paper by Yotov et al. [24] shows that both empirical and model-driven methods of code optimization lag behind the performance that can be obtained through careful hand optimization. At present, the reasons for this performance gap are not well understood [24]. Choosing the best set of optimizing transformations is not easy because transformations interact in complicated ways. Also, the best parameterization for a collection of transformations is highly dependent on the code to which they are applied.

Tailoring large applications to modern architectures by manually restructuring them is often impractical. Not only is the tailoring process labor intensive, but after transformations such as unrolling, blocking and iteration set splitting are applied, codes become difficult to understand and maintain. To address this problem, we have developed a source-to-source transformer that enables application scientists to achieve high performance with programs written in Fortran 77, yet maintain their codes in a natural style. To improve performance, developers augment their code with directives to control the application of optimizing transformations including multi-level loop fusion, multi-level blocking, unroll-and-jam, and iteration space splitting. When our tool is run on a program augmented with directives, it checks the legality of the optimizations specified by the directives, performs the transformations requested and then outputs the transformed code in source form.

The principal novelty of our tool is its integrated support for a set of sophisticated transformations and the precise control it gives users over how they are applied. The information from directives is passed not only to the corresponding transformation but also to other interacting transformations that require it. Information is also passed between transformation modules (e.g from loop alignment to fusion and storage reduction). Each transformation has been tailored so that transformations can be applied in concert.

We describe preliminary experiments using our tool to tune the performance of three comptuational kernels: the Runga-Kutta advection core from the NCOMMAS code for mesoscale weather modeling, Livermore Loop 18 and the swim code from SPEC 2000 benchmark. By using our tool to apply multiple transformations in concert, we were able to double the performance of each of the codes on an SGI O2 workstation.

In the sections that follow, we review related work, describe our tool's integrated transformation framework along with key analysis and optimizations it provides, present results of preliminary experiments applying our tool, and briefly discuss our conclusions and future plans.

2 Related Work

Loop fusion and loop blocking have been studied extensively [10, 11, 23, 7, 9, 18]. Fusion reduces reuse distance of a particular data by merging two iteration spaces. Blocking reshapes an iteration space over a data domain by partitioning it into tiles that fit comfortably into cache and then completing all computations on each tile before moving to the next. Unroll-and-jam has been previously noted as a useful transformation for reducing pipeline interlocks [5] and exploiting outer loop temporal reuse [4]. We selectively unroll-and-jam imperfect loops and interleave statements from jammed loop bodies to reduce register lifetimes and improve ILP.

Many researchers have studied combining loop transformations with transformations for reducing the storage of array temporaries [12, 15, 21, 16, 25, 20]. Lewis et al. [15] proposed using array level analysis and apply loop fusion and array contraction directly to array statements for languages using array syntax. Gao et al. presented a method called collective loop fusion [12]. They first partitioned a collection of loop nests into fusible clusters using max-flow min-cut algorithm. Loop fusion and array contraction into scalars are performed on all loop nests within a cluster. They only considered fusion of conformable loop nests and do not apply loop alignment to remove fusion-preventing dependences.

Lim et al. used affine partitioning to support blocking and array contraction [16, 17]. Arrays are contracted to scalars or lower-dimensional arrays within each independent thread and expended by a block size when blocking is applied to interleave the threads. Our storage reduction framework is more general and it is also integrated with other transformations in our framework.

Song et al. presented a network flow algorithm which provably minimizes memory requirement for multi-dimensional cases [21]. To achieve that, loops are coalesced together into single-level loops before loops are fused and after loop shifting is applied. Loop trip counts are assumed to be equal at the same corresponding loop level, otherwise loops need to be prepeeled and partitioned. However, coalescing may introduce more conditionals and subscript indexing overhead. They use a heuristic to avoid register spilling and excess cache misses to avoid over-fusion. Our storage reduction is not to minimize memory requirement, but rather reduce it to nearly minimum and generate efficient code as well.

Pike and Hilfinger [20] also exposed transformation parameters for tuning. In their work, they combine loop fusion and tiling with array contraction in their Titanium compiler. However, their fusion and tiling only apply to a set of consecutive `foreach` loops each containing a single statement. Our techniques apply to more complex programs and include a more comprehensive set of transformations.

A recent paper by Ghosh et al. [13] discusses the high level optimizer (HLO) developed by Intel for the Itanium architecture. The main difference between HLO and our tool is that HLO does not expose any of the

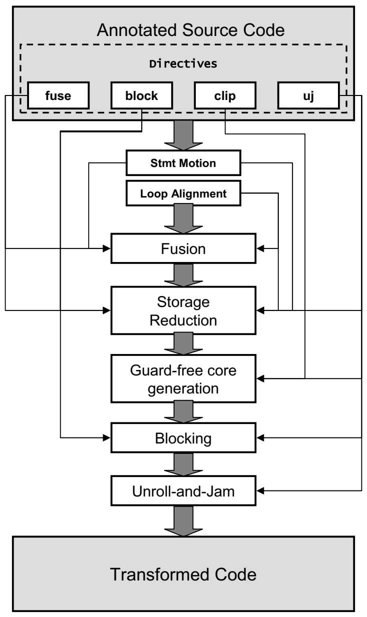


Figure 1: Overview of transformations.

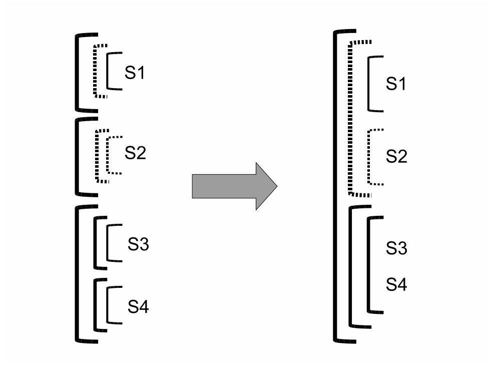


Figure 2: Controllable multi-level loop fusion.

transformation parameters for external control. We directly compare the effectiveness of our tool with HLO in Section 5.

Our tool uses the Omega library to represent data and iteration sets [14]. Omega enables us to perform complex transformations relatively easily and facilitates the integration of transformations in our framework.

3 Approach

Although there has been considerable recent research in improving program performance using both model-guided and empirical methods, at present these approaches fall short of hand-coded performance for general cases. Complex processor architectures and the complex interaction of transformations make selecting the best set of transformations with the best parameters very difficult. As a result, most application programmers either settle for sub-optimal performance or they try to manually tune their applications. Manual tuning of programs on today's modern microprocessor based architectures can be an arduous task if the programmer also has to perform the transformations by hand. With this in mind, we designed a tool containing a sophisticated set of transformations that can be useful for manual tuning of programs. At the same time we gave the user fine-grain control over these transformations which can help her get the desired performance from her programs. Since, any automatic method is unlikely to give the best performance for a particular program, we believe having control over the transformation parameters will allow the application programmer get the desired performance from the set of programs she wants to tune.

Paul Woodward and his team in the Laboratory for Computational Science and Engineering (LCSE) at the University of Minnesota have considerable (painful) experience in manually restructuring sophisticated applications to achieve high performance on microprocessor-based systems. Through a partnership forged as part of NCSA's Performance expedition, at Rice University we set out to build a source-to-source transformer that automates key transformations employed by the Minnesota team. Using Rice University's D System compiler infrastructure, we developed a tool for performing source-to-source transformations of Fortran 77 programs. Our tool enables application scientists to maintain codes in their natural form, yet achieve high performance by annotating the codes with directives that indicate how the tool should restructure them to improve their efficiency.

Our program transformation tool enables complex optimizations to be applied in concert to achieve performance that was previously only achievable through careful hand optimization. Figure 1 gives an overview of our framework and shows the tight integration among the various transformations. We start off with source code annotated with directives that specify the parameters for the transformations. The thin arrows in the figure represent the flow of information from directives to the transformation modules and from one module to another. The thick arrows show the sequence in which the transformations are applied. The key to the success of this approach is to make each transformation aware of all other related transformations that are being applied. Most information is derived from directives in the source code and is available to each transformation module irrespective of the order in which transformations are applied. Because the transformations are closely related, having relevant information about other transformations makes each transformation potentially more effective. Below we illustrate key interactions between transformations as we discuss using transformations to improve program performance.

Data-intensive scientific codes suffer from poor memory hierarchy utilization. Reuse can be improved by transforming the code to reduce temporal reuse distance. Our tool employs *multi-level loop fusion* to reduce temporal reuse distance in programs composed of multiple loop nests. Our tool supports selective loop fusion at any or all levels, as we explain in Section 3.1. When a set of fused loops contains both the definition and all uses of a temporary array, it is often possible to apply a *storage reduction* transformation to reduce the rank of the array and/or the extent of one or more array dimensions. In some cases, an array can be contracted to a scalar. We use a *loop alignment* transformation to offset iteration spaces with respect to one another to enable fusion by preserving def-use ordering (i.e. respect to true data dependences) and to improve storage reduction by shortening the dependence distances. To further improve temporal reuse, we apply a *blocking* transformation to loops. When a loop is blocked, it can create an opportunity to further reduce storage by enabling the extent along a blocked dimension of a temporary array to be reduced.

When multiple iteration spaces with overlapping but not identical loop ranges are fused, the resulting iteration space may not lead to efficient code if there are conditionals in inner loops; this could be especially problematic for software pipelining on processors that lack support for predicated execution. We developed a *loop splitting* transformation that partitions an iteration space into three segments: a prologue, a core, and epilogue. The core consists of the loop range in which all statements execute. The prolog and epilog consist of parts of the iteration space where some, but not all of the statements are active. To improve pipeline utilization as well as to create opportunities exploiting outer-loop we employ *unroll-and-jam*. For unroll and jam to be efficient, the range of the unrolled loop must be a multiple of the unroll factor. Typically, unroll-and-jam creates a prolog and/or epilog that deal with extra iterations. In our integrated tool, the loop splitting transformation ensures that the range of the core loop is a multiple of the unroll factor; thus, it integrates prolog and epilog generation for fusion with that for unroll-and-jam.

In the subsequent sections, we describe the key analyses and transformations supported by our tool along with their interactions in a bit more detail.

3.1 Controllable multi-level loop fusion

In programs in which the computation is decomposed into a sequence of simple loop nests, opportunities for temporal and/or spatial reuse of data in cache are often missed when operating on large data sets. In such programs, one loop will often use or define values (or in the case of spatial reuse, cache lines) needed by a later loop nest only to have them evicted from cache before that reuse is realized. To capitalize on such opportunities for reuse, loop fusion is needed to bring multiple uses of values (or cache lines) closer together temporally.

When an array of multiple dimensions is manipulated by a program in several loop nests, there are several ways in which these loop nests could be fused. Consider a pair of loop nests manipulating at least one common three-dimensional array. For a moment, suppose there are no dependences preventing fusion between any of the corresponding loops in the two nests. In this case, the outermost one, two, or all three loops could be fused to shorten the distance between uses of values, cache lines, or TLB entries. It might seem that fusing as many loops as possible would yield the best performance since temporal reuse distance would be shortest; however, fusing all loops could result in a complex inner loop body that suffers too many conflict misses. In cases where dependences prevent fusion at all levels, fusing on one or more levels may still be useful if the data arrays manipulated are not enormous.

To enable an application programmer to precisely control how fusion is applied to a series of loop nests in a program, we provide a *fuse* directive. By default, each loop is assigned a unique fusion group identifier and will not be fused with any other loop. The *fuse* directive can be used to assign a particular fusion group id to any loop. A *fusion control tuple* for a loop consists of a vector (whose length corresponds to the loop depth) containing the fusion group id for the loop itself and each enclosing loop. Fusion group ids in a fusion control tuple are organized according to the depth of the loop that they represent in ascending order. If two loops have the same k elements in their fusion control tuples, fusion is legal and there are no intervening statements in a different fusion group, then they are fused. Figure 2 shows a schematic representation of several loop nests before and after controlled fusion. Each loop is represented as an arc indicating its scope. On the left half of the figure, the line style of each arc represents the fusion group id for a loop. The fusion control tuple for the inner loop surrounding statement `s1` consists of a vector of line styles [*solid, dashed, solid*]. On the right, we see that corresponding loops with equivalent fusion control tuples have been fused.

3.2 Statement motion

In scientific applications, loops can be arbitrarily nested and statements may exist inside or outside loops. Unrelated statements between two fusible loops are an impediment to simple fusion. Intervening statements can always be embedded in the fused loop as long as they don't lead to fusion-preventing dependences; however, this approach can be inefficient, especially if conditionals need to be added around the embedded statement. To avoid wanton embedding when scalar assignments separate two loops, we implemented a statement motion transformation. To perform this transformation, we first examine control and data flow constraints that affect the statement. Subject to these constraints, we hoist the statement out of as many loop levels as possible and move it to its earliest possible position in the code. [1]

```
[s1]   c1 = 1.0
       do j = 4, 99
         do i = 2, 97
           q(i,j) = p(i+2,j-2)
[s2]     c2 = j
         do i = 2, 97
           r(i,j) = c2*q(i,j)
[s3]   c3 = c1+2.0
       do j = 2, 99
[s4]     c4 = 4.0
         do i = 2, 99
           p(i,j) = c3*r(i,j)+c4
```

(a) Before statement motion

```
[s1]   c1 = 1.0
[s3]   c3 = c1+2.0
[s4]   c4 = 4.0
       do j = 4, 99
[s2]     c2 = j
         do i = 2, 97
           q(i,j) = p(i+2,j-2)
         do i = 2, 97
           r(i,j) = c2*q(i,j)
       do j = 2, 99
         do i = 2, 99
           p(i,j) = c3*r(i,j)+c4
```

(b) After statement motion

Figure 3: An example of statement motion.

Figure 3 shows an example of statement motion. [2] It shows how statements `s1`, `s2`, `s3`, and `s4` are moved by our implementation. Note that `s4` is hoisted out of a loop. After statement motion, our tool can align and fuse both the `i` and `j` loops.

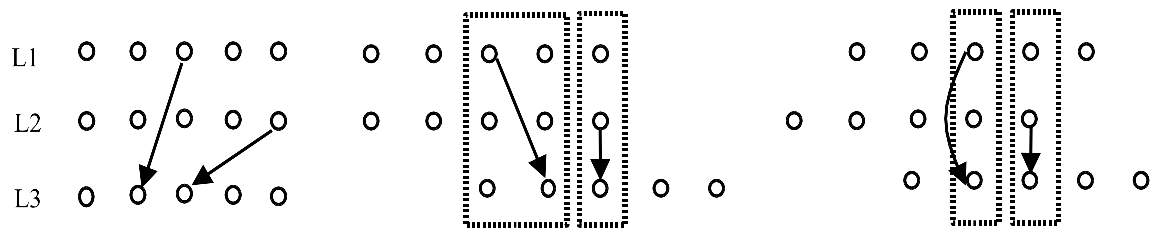
3.3 Necessary and sufficient alignment

Loop alignment has been previously used to enable fusion by shifting statements in different loops relative to a common iteration space [1]. In this section, we describe a *necessary and sufficient alignment* strategy that we implemented in our tool to enable multi-level fusion and reduce storage as well. The algorithm first traverses the loop nests in forward program order to compute a *sufficient* distance that must be maintained between pairs of dependent variables to preserve the program semantics. It then traverses the loops in backward program order to compute the *necessary* distance that needs to be maintained between dependent variables without violating the existing dependences.

Figure 4 shows the applications of alignment with sufficient distance only and with necessary and sufficient distance. In Figure 4(a), there are three neighboring loops `L1`, `L2`, and `L3`. Circles represent iterations and arrows

[1] While there are certainly more thoughtful techniques for moving statements, such as considering *fuse* directives before selecting final placement, we have found this approach sufficient to date. We plan to extend our implementation to move other kinds of statements besides scalar assignments.

[2] For brevity, we omit `enddos` and `endifs` in the code examples shown. Instead, we use indentation to show the code structure.



(a) Before alignment (b) Alignment with sufficient distance (c) Alignment with necessary and sufficient distance

Figure 4: Necessary and sufficient alignment.

represent dependences between the loops. Because of the dependences from `L1` and `L2` to `L3`, `L3` cannot be fused with `L1` and `L2`. However, by applying an alignment with a sufficient distance to `L3`, we can fuse all three loops.

After fusion, the dependence distance between `L1` and `L3`, and `L2` and `L3` are 1 and 0 as shown in Figure 4(b). Assume dependence from `L1` to `L3` is carried on array `a` and dependence from `L2` to `L3` is carried on array `b`. Both `a` and `b` are temporary arrays and the dependences are the only ones carried on them. In that case the size of `a` and `b` can be reduced to two and one element respectively. This is indicated by the width of the grey dashed boxes in Figure 4(b). On the other hand, if we apply an alignment with a necessary and sufficient distance, that it, we shift `L1` one iteration right as well, the result after the alignment is shown in Figure 4(c). Since the distances of both dependences are 0 now, one element for each of `a` and `b` will suffice. Although the effect of the necessary and sufficient alignment on storage reduction is not significant in this case (from three elements to two), the purpose of this example is to show the main idea. When the temporary arrays are high dimensional or the storage reduction is applied in conjunction with transformations such as blocking and unroll-and-jam, the reduction could be substantial.

Necessary and sufficient alignment is also beneficial for guard-free core generation by enlarging the core and reducing the number of clippied tiles.

3.4 Blocking

Blocking (also known as *tiling*) is widely recognized in the literature as an effective technique for improving cache utilization for temporal reuse. Our tool enables a developer to mark any loop with a *block* directive to specify an integer blocking factor for that loop. For blocking to be legal, it must be legal to interchange each blocked loop with one or more enclosing loop in the nest. The alignment transformation performed for fusion guarantees that this precondition for blocking is satisfied. Although blocking is an effective transformation on its own, having it integrated with other transformations adds to the overall effectiveness of our tool. In particular, there is strong interplay between blocking and storage reduction.

Consider the case, where we have a three-dimensional array, and a true dependence of distance d involving the array carried on the outermost k loop of a three dimensional loop nest; the two inner loops iterate over dimensions i and j. Using the storage reduction strategy described in Section 3.7, a full size temporary is not needed for this computation because only a slab of thickness d contains live values along the k dimension. However, if we block the i and j loop nests using blocking factor b_i and b_j, the storage can be reduced to a volume of $b_i \times b_j \times d$. In our framework, all blocking information is communicated to the storage reduction algorithm. This enables us to reduce storage in multiple dimensions of a temporary array whenever the opportunity arises.

3.5 Guard free core generation

As mentioned previously, the presence of conditionals within a fused loop nest can degrade performance on a pipelined architecture. We developed a novel loop splitting algorithm that partitions the iteration space in a way that enables us to have a core loop nest free of conditionals. Our algorithm is designed for optimizing a loop nest created by fusing a series of stride-one rectangular iteration spaces that are offset with respect to one another.

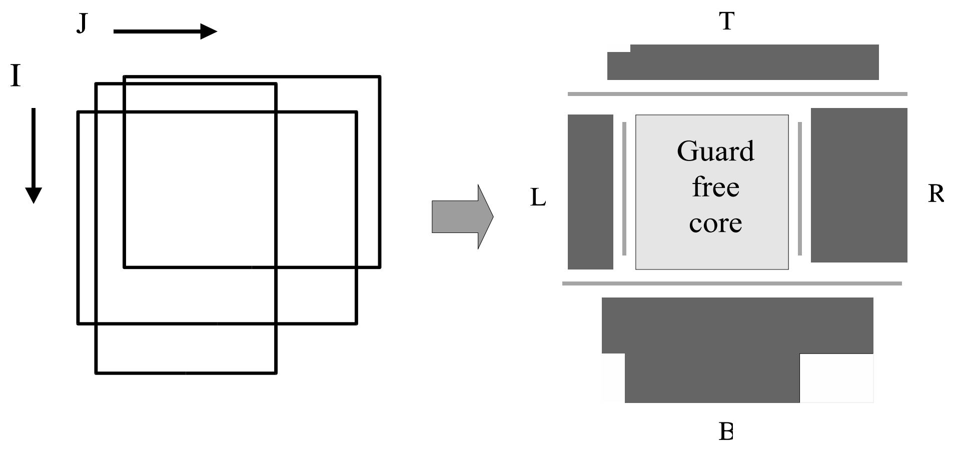


Figure 5: Guard-free core computation.

We begin our loop splitting process by first computing a common intersection region (the *core*) of all the overlapping iteration spaces of the fused loop nest. By performing a series of projections and intersections, we whittle this overlapping iteration space into a hyper-rectangular core. Next, we clip the core so that the range of each dimension is a multiple of any unroll factor that will be applied along that dimension as part of a later unroll-and-jam transformation. This clipping transformation is applied to simplify the subsequent application of unroll-and-jam to the loop nest. Finally, we use half space projections aligned with surfaces of the hyper-rectangular core to clip off prolog and epilog iteration spaces around the core along the dimensions specified by the *clip* directive. We apply this clipping transformation to each statement in the fused loop nest. The union of all the resulting iteration spaces gives us the clipped region for a particular dimension. For each dimension, we end up with two such clipped regions depending on the type of halfspace - *top* or *bottom* - we consider. The guard-free region for the dimension is obtained by taking the intersection of the top and the bottom halfspace. Figure 5 illustrates this transformation. The figure on the left shows the overlapping iteration spaces whereas the figure on the right shows the guard-free core in which all iteration spaces overlap, surrounded by clipped wings.

3.6 Unroll-and-jam

Unroll-and-jam is a well-known transformation that has been shown to improve pipeline performance [5], register usage and cache behavior [6]. The transformation involves unrolling an outer loop a specified number of times and then jamming the resulting copies into the innermost loop. This transformation brings the iterations of the outer loop closer together to exploit temporal reuse carried by the outer loop. Moreover, by duplicating the loop body, unroll-and-jam exposes more instruction level parallelism (ILP).

Studies have shown that unroll-and-jam is not always beneficial. Register spills can occur if the register pressure of the unrolled loop body is greater than the available number of registers [6]. In some cases the unrolled loop body could become large enough to thrash a small I-cache. For large loop bodies, we observed that unroll-and-jam may not increase register reuse or ILP because the jammed instances are too far apart in the unrolled loop body and the back end compiler can't determine that they can be interleaved safely.

These issues with unroll-and-jam become especially significant in our integrated framework. Application of statement motion and multi-level loop fusion can create loop nests with a large inner-loop body. Moreover, fusing multiple loops is likely to cause disjoint reuse patterns to appear in the fused loop nest. These factors can have a negative impact on the effectiveness of the unroll-and-jam transformation.

Our implementation of unroll-and-jam incorporates two refinements that address the problems we outlined above. The first strategy, *selective unroll-and-jam* enables finer control over which statements within a fused loop should be unrolled. This strategy reduces the size of the unrolled loop body. Also, this eliminates needless jammed instances of statements within an unrolled loop and prevents dilation of reuse distance for reuses carried by the inner loops. This can potentially improve cache behavior. The second strategy, *statement interleaving* is targeted to get more ILP and better register usage when we have a large unrolled loop body. The two strategies

```
      do j = 2, 1001
        do i = 2, 1001
          do k = 1, 1000
[s1]        ab(k,i,j) = a(k,i,j-1)+a(k,i,j-2)
          do k = 1, 1000
[s2]        al(k,i,j) = b(k,i-1,j)+b(k,i-2,j)
```

(a) Before selective unroll-and-jam

```
      do j = 2, 1001, 2
        do jj = j, j + 1
          do i = 2, 1001
            if (jj .eq. j) then
              do k = 1, 1000
[s1]            ab(k,i,j) = a(k,i,j-1)+a(k,i,j-2)
                ab(k,i,j+1) = a(k,i,j)+a(k,i,j-1)
            do k = 1, 1000
[s2]          al(k,i,jj) = b(k,i-1,jj)+b(k,i-2,jj)
```

(b) After selective unroll-and-jam

Figure 6: Example of selective unroll-and-jam.

are described below in some detail.

3.6.1 Selective unroll-and-jam

Our framework gives the user control over which loops and which statements within fused loops should have unroll-and-jam applied. Through the use of directives the user can select individual statements within a fused loop and specify the unroll-and-jam amount for that particular statement. This high level of control can significantly improve the effectiveness of the unroll-and-jam transformation by reducing the number of capacity and conflict misses.

Consider the code fragment in Figure 6(a). Suppose, we decide to unroll-and-jam the outermost loop to exploit the reuse carried by `j`. If we unroll-and-jam both `s1` and `s2` in `j` we will end up with four statements in the body of loop `i`. We notice, however that the reuse in `j` only applies to `s1`. The reuse in `s2` is carried by `i`. So, having a jammed instance of `s2` is not helpful. In fact, having `s2` jammed unnecessarily dilates the reuse distance carried by `i` increasing the likelihood of capacity misses. Moreover, since `s1` and `s2` refer to two different arrays - `a` and `b` - this situation can potentially increase conflict misses. Both of these factors can degrade performance.

We can circumvent this problem by unroll-and-jammming selectively. As Figure 6(b) illustrates, this can be achieved by wrapping the loop body of the unrolled loop inside a new loop `jj` that iterates the number of times the outer loop has been unrolled. Here, we use a conditional to embed the jammed instances of `s1` into the first iteration of loop `jj`. This has the effect of unroll-and-jamming `s1` in `j` while leaving `s2` unchanged. This yields the desired register reuse from `s1`. At the same time, it avoids unnecessarily perturbing the cache reuse carried by `i` for `s2`.

3.6.2 Statement interleaving

When jamming an unrolled loop body into the innermost loop, we interleave the statements of the two loop bodies whenever it is legal to do so. Keeping the jammed instance of a statement close to the original instance facilitates close temporal reuse by reducing the register lifetime of values and also boosts ILP since the statement instances from the separate loop iterations are generally independent.

3.7 Storage reduction

As described in Section 1, calculations in complex scientific applications are often factored into a series of steps with partial results carried from loop to loop by short-lived temporary arrays. However, using large temporary arrays can hurt program performance on modern microprocessor-based systems. Automatically restructuring a code to reduce the size of temporary arrays used can significantly improve performance by reducing the code's need for memory bandwidth, improving cache utilization, and reducing the data footprint in TLB. In this section, we describe our tool's strategy for performing this optimization as part of an integrated suite of program transformations.

For storage reduction to be possible, the entire live range of an array temporary (a local variable in a procedure) must be inside the scope of a single (perfect or imperfect) loop nest. Loop fusion can dramatically increase the opportunities for applying this optimization. After fusion, the extent of a temporary array along a dimension

```
thirddtbydx = dt/(3.*dx); halfdtbydx = 1.5*thirddtbydx
thirddtbydz = dt/(3.*dz); thirddtbydy = dt/(3.*dy)
do j = -5, ny+7
  do i = max(-j-9,-5), nx+7
    if (i .ge. -4) then
      do k = -5, nz+6
        ab$(k,i-1,iand(j,1)) = (f60*(a(k,i-1,j-1)+a(k,i-1,j))+ &
          f61*(a(k,i-1,j-2)+a(k,i-1,j+1))+f62*(a(k,i-1,j-3)+a(k,i-1,j+2)))*...
    if (j .ge. -4) then
      do k = -5, nz+6
        al$(k,iand(i,1)) = (f60*(a(k,i-1,j)+a(k,i,j))+ &
          f61*(a(k,i-2,j)+a(k,i+1,j))+f62*(a(k,i-3,j-1)+a(k,i+2,j-1)))*...
    if (i .ge. -4 .and. j .ge. -4) then
      do k = -5, nz+7
        af$(k) = (f60*(a(k-1,i-1,j-1)+a(k,i-1,j-1))+ &
          f61*(a(k-2,i-1,j-1)+a(k+1,i-1,j-1))+f62*(a(k-3,i-1,j-1)+a(k+2,i-1,j-1)))*...
    ......
```

Figure 7: Multi-level loop fusion and storage reduction.

indexed by a fused loop can often be reduced. The minimum extent of each dimension of a temporary array depends principally on the distances of associated data dependences and the nesting level of loops carrying these dependences. A dimension can be eliminated when its extent can be reduced to one.

When storage reduction is to be applied in concert with loop transformations, support for all transformations must be carefully integrated to achieve the desired effect. Our program transformation tool enables detailed specification of the desired shape of loop nests through directives that specify fusion groups, blocking factors, unroll-and-jam factors and whether iteration space splitting should be applied. Each of these transformations interacts in a significant way with automatic storage reduction. The offsets chosen when aligning loops for fusion can affect dependence distances in the fused loop nest and thus the extent to which storage can be reduced. Loop blocking can reduce the extent needed along a dimension of a temporary array. Unroll-and-jam can diminish the storage reduction possible along an array's dimension indexed by a jammed loop index. Finally, reducing storage along a dimension can make it impossible to split the iteration space of a loop whose index is used along that dimension.

In our tool, storage reduction uses information from each of the aforementioned loop transformations to infer the shape of the code that will result and reduces storage as to the greatest extent possible (as dictated by data dependences and loop nesting order) with the additional constraint that it must be possible to efficiently index the reduced storage. When the extent of an array dimension carrying one or more data dependences is reduced, the indexing scheme in that dimension must be *wrapped* so that indexing repeatedly cycles through the data of reduced extent. Indexing is most naturally wrapped using a `mod` operation. However, computing `mod` is expensive and the cost of computing it to wrap an array dimension can outweigh the potential performance benefits of reducing storage. When the extent along a reduced dimension of an array is small, the LCSE team at University of Minnesota has found that rotating a set of subscripts through a set of scalars can dramatically reduce instruction count for wrapped indexing. However, this strategy increases integer register pressure and can cause register spills. Currently, our tool generates code using the Fortran `iand` intrinsic (a bitwise logical and) to wrap indexing. The `iand` operation is cheaper to compute than `mod`; however, using it requires that the extent of a reduced dimension be a power of two. This can limit our ability to reduce storage. In practice, we have found our `iand` based indexing scheme enables significant storage reductions and yields fast code.

The code in Figure 7 shows a portion of the original `twostages` and an optimized version after applying storage reduction to a loop that has been fused at the two outermost levels. Prior to storage reduction, the code used three full-size, three-dimensional array temporaries temporaries `ab`, `al` and `af`. The code in the Figure 7(b) shows the storage-reduced version in which `ab` has been replaced with `ab$` which has only two planes in the `j` dimension, `al` has been replaced with `al$` which has been reduced to two columns, and `af` has been replaced with a one-dimensional array `af$`. The `k` dimension was not reduced for any of the arrays in this example because only the two outer loops are fused.

In the future, we expect to enhance our tool to eliminate all arithmetic overhead for wrapped indexing (when the extent of a wrapped dimension is small) by unrolling code for the core computation and replacing subscripts in the reduced dimension with constants. For this approach to work, one must unroll by a multiple of the length of the reduced dimension, add a loop prolog necessary to establish the pre-conditions for the unrolled code and an epilog to handle remaining iterations.

4 Experimental Results

To show the effectiveness of our program transformation tool, we present experimental results of the Runga-Kutta advection kernel (`twostages`) from the NCOMMAS code for mesoscale weather modeling [22], the Livermore Loop 18 benchmark (`LL18`), an explicit hydrodynamics kernel, and the `swim` benchmark from the SPEC 2000 suite. `twostages` has 116 lines of Fortran code with 8 three-level loop nests using 15 three-dimensional real arrays including globals and temporaries. `LL18` is a 59-line kernel with 4 two-level loop nests using 9 two-dimensional double precision arrays. We report results on two problem sizes for each program. Problem sizes for `twostages` are $64 \times 64 \times 64$ and $128 \times 128 \times 128$, plus a ghost region with extra nine elements at each end of the data dimensions. Problem sizes for `LL18` are 512×512 and $1K \times 1K$. Problem sizes for `swim` are 256×256 and 512×512.

Performance experiments were conducted on an SGI O2 workstation with a 195 MHz MIPS R10K processor. The processor has a 32KB 2-way set associative primary cache, a 1MB 2-way unified secondary cache, and a 64-way 512KB TLB. Each version of the transformed code was generated by our tool based on annotated original source code. All programs were compiled with MIPSpro compiler V7.3.1 using "-O3 -mips4" optimization flags. We used SGI's SpeedShop tool to measure CPU cycles, L1, L2, and TLB misses using hardware performance counters. All measurements were taken on separate runs to avoid multiplexing the performance counters. Each experiment was repeated multiple times. Variations were small and the average measurements are reported. All measurements were taken for 2 time steps for each `twostages` version and 5 time steps for each `LL18` version.

We compare the performance of the original code with different versions of transformed code generated by our tool. A different combination of transformations was used for each separate run of the experiment. Statement motion, fusion, and loop alignment were applied to all versions of the transformed code since they are enabling transformations for others in our framework. Loop splitting was used whenever unroll-and-jam was used. The parameters for fusion, blocking and unroll-and-jam were also varied in turn. For the `swim` benchmark a naive scalar replacement(`SC`) transformation was used.

For each version of the transformed code we present overall speedup as well as the ratio of L1, L2 and TLB misses relative to a baseline version of the code. For the baseline version, we also present the total number of L1, L2, and TLB misses. In each table, the leftmost column describes the transformations that were applied. We use the notation `n-lF` to denote that an n-level fusion was applied to the source code. `BL, SR, UJ` are abbreviations for blocking, storage reduction and unroll-and-jam respectively. So, for example, `2-lF, SR, BL` implies that in the transformed code, the outer two loop levels were all fused, then both storage reduction and blocking were applied.

In Table 1– 3 we present results obtained using different sets of transformations and compare them to the baseline version of the code. The results show that we are able to double the performance for both `twostages` and `LL18` by using the *best* combination of transformations. For `twostages` this involved applying all four of the major transformations in our framework. For `LL18` a combination of storage reduction, blocking and fusion turned out to be the best.

For `swim` the improvments were less dramatic. A combination of unroll-and-jam, blocking and scalar replacement yielded a 24% speedup. It should be noted that even after the improvement `swim` was performing at only about 7% of peak FP performance. This suggests that there may be more opportunities for improving performance that are still available.

The results in Table 1– 3 also reveal the contribution of each transformation in achieving the best performance. We see that eliminating any one of the transformations from the *best* combination results in a loss of performance. That is, each transformation is necessary to obtain the best results.

5 A Comparison with Intel's High Level Optimizer

To support our case for using a capable transformation tool suitable for user guided tuning, we present here a comparison of our tool with the high level optimizer (HLO) released by Intel for the Itanium architecture [13].

Code	*64 × 64 × 64*				*128 × 128 × 128*			
	L1 misses	*L2 misses*	*TLB misses*	*Speedup*	*L1 misses*	*L2 misses*	*TLB misses*	*Speedup*
`2-lF,SR,BL,UJ`	0.86	0.40	0.29	1.99	0.86	0.42	0.40	1.91
`2-lF,SR,BL`	1.00	0.39	0.30	1.90	1.01	0.43	0.41	1.79
`2-lF,BL,UJ`	1.27	0.58	0.49	1.49	1.15	0.61	0.61	1.47
`2-lF,SR,UJ`	0.81	0.46	0.21	1.89	0.82	0.74	0.29	1.34
`Original`	1(3.09M)	1(1.38M)	1(134K)	1	1(23.3M)	1(8.5M)	1(540K)	1

Table 1: Contribution of individual transformations in achieving best performance for `twostages`.

Code	*512 × 512*				*1K × 1K*			
	L1 misses	*L2 misses*	*TLB misses*	*Speedup*	*L1 misses*	*L2 misses*	*TLB misses*	*Speedup*
`2-lF,SR,BL`	0.57	0.42	5.95	2.12	0.48	0.42	24.1	1.96
`2-lF,SR`	1.01	0.38	0.44	1.94	0.96	0.44	0.53	1.82
`2-lF,BL`	0.93	0.65	18.9	1.34	0.92	0.65	45.12	1.18
`Original`	1(8.2M)	1(2.1M)	1(25K)	1	1(39M)	1(8.6M)	1(125K)	1

Table 2: Contribution of individual transformations in achieving best performance for `LL18`.

HLO integrates a large set of source-to-source transformations that exploit the available ILP in programs. It includes most of the major transformations available in our tool such as *fusion, array contraction* and *blocking.* In addition HLO also has other optimizing transformations such as *data prefetching*, *scalar replacement*, etc. The chief difference between Intel's HLO and our framework is that HLO does not allow the user to tune any of the transformation parameters. Also the choice of transformations that the user can enable/disable for a particular compilation is very limited.

To compare the two tools we first compiled the `twostages` kernel with Intel's fortran compiler using '-O3' option. The '-O3' option applies the most aggressive transformations and enables all transformations in HLO. We used the *opt_report_file* option to generate the list of high level transformations that were applied. We then ran the twostages kernel through our tool several times applying a new transformation for each run. After running the program through our tool we compiled it using the '-O3' option again and generated a report of all optimizations that were applied. We also collected performance metrics for each version of the code.

Table 4 lists the transformations that were applied by our tool and HLO at each step. Table 4 also lists the memory peformance metrics and the overall speedup for each version of the transformed code.

We see from the table that HLO did not apply fusion, array contraction or blocking to the code, although all these transformations proved beneficial when applied by our tool. The cumulative effect of using our tool to apply these transformations was a 31% improvement over what HLO could achieve alone. There are three possible explanations for HLO not applying the transformations in question. First, it may be that the opportunities of applying fusion, array contraction or blocking were not detected because of the particular code shape. Secondly, it may be that the transformations supported by HLO were not sophisticated enough to be applied in this case. For example, our tool uses both statement motion and loop alignment to enable fusion for the loops in the kernel. If such enabling transformations were not used the loops in the kernel would not be fusable. Since no details are given in the literature [13] about the precise capabilities of HLO's transformations, we were unable to determine if this indeed is the case. Lastly, it is also possible that opportunities *were* detected but the analysis in HLO deemed those transformations to be unprofitable. In any case, the results presented here support our argument that using directives to guide the optimization process is a necessary step in getting close to hand coded performance.

6 Conclusions and Future Work

Given the enormous complexity of today's processor architecture, building exact architectural models have become an intractable task. As such most optimizing compilers use simplified models that have obvious short-

Code	256 × 256				512 × 512			
	L1 misses	*L2 misses*	*TLB misses*	*Speedup*	*L1 misses*	*L2 misses*	*TLB misses*	*Speedup*
`UJ,SC,BL`	0.98	0.96	0.46	1.21	0.93	0.98	1.01	1.24
`SC,UJ`	0.97	0.93	0.54	1.19	0.93	0.99	1.06	1.21
`SC,BL`	0.98	0.95	0.52	1.17	0.95	1.02	1.04	1.20
`UJ,BL`	0.99	0.97	0.69	1.01	0.92	1.03	0.97	1.09
`Original`	1(5M)	1(2.2M)	1(485K)	1	1(22.1M)	1(8.4M)	1(1.4M)	1

Table 3: Contribution of individual transformations in achieving best performance for `swim`.

Code	*Transformations applied by HLO*	*L2 misses*	*L3 misses*	*TLB misses*	*Speedup*
`Original`	Prefetch	1(127M)	1(86M)	1(1.08M)	1
`2-lF`	Prefetch, Scalar Replace, Load-pair, Unroll	0.9	1.24	1.09	1.03
`2-lF,SR`	Prefetch, Scalar Replace, Load-pair, Unroll	0.71	0.86	0.67	1.11
`2-lF,SR,BL`	Prefetch, Scalar Replace, Load-pair, Unroll	0.42	0.38	0.94	1.31

Table 4: Comparison of the transformation tool with Intel's HLO on the `twostages` kernel.

comings. An empirical method can overcome some of these shortcomings by searching for the best transformation parameters through iterative evaluation of the program. However, the optimization space is large and complex and searching this space effectively and efficiently is a subject of ongoing research by ourselves and others [24, 2, 8, 19].

Until the time compilers are able to automatically generate code that is comparable to state-of-the-art hand optimized code, the semi-automatic approach of the transformation tool described in this paper provides a good alternative. As evidenced in the experimental results of Section 4, the tight integration of transformations in our tool and the ability to manually tune the optimization parameter has been key in achieving high performance. Our tool enables application programmers to maintain their code in a natural style and at the same time provides the kind of performance that leading edge commercial compilers are unable to provide at present. It is also noteworthy that our source-to-source tool is suited for generating tailored code for stencil calculations [3].

Our plans for the future is to employ our tool as a back-end for a model-guided empirically-driven optmization system. We believe that our tool can serve a key role in such a system by faithfully executing a complex set of transformations in an integrated fashion.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, 2002.

[2] N. Baradaran, J. Chame, C. Chen, P. Diniz, M. Hall, Y.-J. Lee, B. Yu, and R. Lucas. ECO: an empirical-based compilation and optimization system. In *International Parallel and Distributed Processing Symposium*, 2003.

[3] M. Bromley, S. Heller, T. McNerney, and G. Steele, Jr. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[4] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[5] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358, Aug. 1988.

[6] 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.

[7] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, San Jose, CA, Oct. 1994.

[8] B. Childers, J. Davidson, and M. L. Soffa. Continuos compilation: A new approach to aggressive and adaptive code transformation. In *International Parallel and Distributed Processing Symposium*, 2003.

[9] S. Coleman and K. S. M^{c}Kinley. Tile size selection using cache organization. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[10] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[11] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *International Parallel and Distributed Processing Symposium*, San Francisco, CA, Apr. 2001. (Best Paper Award.).

[12] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, Aug. 1992.

[13] S. Ghosh, A. Kanhere, R. Kirshnaiyer, D. Kulkarni, W. Li, Chu-Chow, and J. Ng. Integrating high-level optimizations in a production compiler: Design and implementation experience. In *Compiler Construction: 12th Interantional Conference*, Lecture Notes in Computer Science 2622. Springer-Verlag, 2003.

[14] 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.

[15] E. Lewis, C. Lin, and L. Snyder. The implementation and evaluation of fusion and contraction in array languages. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[16] A. Lim and M. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Snowbird, Utah, June 2001.

[17] A. Lim and M. Lam. Cache optimizations with affine partitioning. In *Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing*, Portsmouth, Virginia, Mar. 2001.

[18] N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(5), 1998.

[19] D. Parello, O. Temam, and J. Verdun. On increasing architecture awareness in program optimizations to bridge the gap between peak and sustained processor performance – matrix-multiply revisited. In *Proceedings of SC'02: High Performance Networking and Computing*, Nov. 2002.

[20] G. Pike and P. N. Hilfinger. Better tiling and array contraction for compiling scientific programs. In *Proceedings of SC'02: High Performance Networking and Computing*, Baltimore, MD, Nov. 2002.

[21] Y. Song, R. Xu, C. Wang, and Z. Li. Data locality enhancement by memory reduction. In *Proceedings of the 15th ACM International Conference on Supercomputing*, Sorrento, Italy, June 2001.

[22] L. J. Wicker. NSSL collaborative model for atmospheric simulation (NCOMMAS). `http://www.nssl.noaa.gov/~wicker/commas.html`.

[23] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[24] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[25] Y. Zhao and K. Kennedy. Scalarizing fortran 90 array syntax. Technical Report CS-TR01-373, Dept. of Computer Science, Rice University, Mar. 2001.