On Reducing Storage Requirement of Scientific Applications

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

Abstract

Over the last two decades, processor speeds have been improving much faster than memory speeds. As a result, memory access delay is a major performance bottleneck in today's systems. Because compilers often fail to automatically choreograph data and computation to avoid memory access delay, we have developed a source-to-source transformation tool for this purpose. To use our tool, developers annotate their code with directives that specify how our tool should apply loop transformations to improve performance. In this paper, we describe a set of storage reduction optimizations that are automatically applied by our tool. These optimizations improve code performance by reducing the memory hierarchy footprint of temporary arrays. Our experiments with a numerical kernel and two weather codes show that our storage reduction optimizations amplify the benefits of loop transformations and doubles performance achievable with loop transformations alone.

1 Introduction

Over the past twenty years, the cycle time of microprocessors has shortened about 60% annually while the cycle time of DRAM has reduced only about 7% per year. To bridge the widening gap between processor and memory speeds, various hardware technologies for improving memory bandwidth and tolerating memory latency, including multiple levels of cache and hardware support for prefetching, have been become commonplace. However, exploiting these technologies most effectively remains a formidable challenge, particularly for complex scientific applications. When the characteristics of a scientific application aren't well matched to the characteristics of the system on which it executes, performance can fall far below the architecture's peak. Furthermore, since different computer systems have different characteristics, achieving the best possible performance can require architecture-specific tuning. For data-intensive applications, improving spatial and temporal data reuse is often the most effective way to boost performance.

For data-intensive programs, loop reordering transformations are commonly used to increase both spatial and temporal reuse [7, 8, 20, 4, 5, 16, 3]. Fusion can reduce reuse distance for data by merging two iteration spaces that access the same locations. Blocking reshapes an iteration space over a data domain by partitioning it into pieces that fit comfortably into cache and then completing all computations on each piece before moving on to the next one. Unroll-and-jam can improve memory hierarchy performance for perfectly nested loops. By unrolling an outer loop and fusing the resulting copies of the loop it contains, one can exploit spatial and temporal reuse at the outer level. Combining unroll-and-jam with scalar replacement (to improve register reuse) is particularly effective.

While strategies for improving spatial and temporal reuse through data and loop transformations have been widely studied by the compiler research community and automatic techniques for such transformations work well on application kernels, their results on real applications often leave much room for improvement. For applications that have rich data reuse over a large program region, achieving good results requires global optimization over the entire region. This case is common in real programs because application developers often 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.

Since automatic techniques for improving spatial and temporal reuse in complex scientific applications often fail to achieve the best possible result, we have developed a tool that enables application developers to choreograph the application of loop transformations to scientific programs. This provides a useful alternative to manually transforming programs when automatic techniques are ineffective. To improve the effectiveness of user-directed loop transformations, we have developed a set of automatic compiler-directed data transformations that reduce the cost of using temporary arrays by shrinking their footprint in a computer's memory hierarchy.

In this paper, we describe a set of compiler transformations that improve temporal reuse in scientific applications by (1) reducing the size of temporary arrays and (2) overlaying storage for multiple temporary arrays that are not live at the same time. We also describe two supporting transformations, statement motion and loop alignment, that improve the effectiveness of storage reduction. All of the transformations we describe here have been implemented in a source-to-source transformer that also applies a rich set of loop transformations including fusion, blocking, splitting, unroll-and-jam, and time skewing. Experiments show that adding our storage reduction transformations to our tool's capabilities improves the performance achievable when applying the tool to scientific programs. Novelties in our work include the following. First, we use two enabling transformations to improve array contraction. Second, we overlay storage for multiple temporary arrays when live range analysis indicates it is safe to do so. Third, we generate efficient code while achieving nearly the minimum storage required.

In the sections that follow, we review related work, describe our storage reduction optimizations along with two enabling transformations, present results of our preliminary experiments, and briefly discuss our conclusions and future plans.

2 Related Work

Loop alignment has been previously used to improve parallelism by transforming loop-carried dependences to loop-independent dependences and enable loop fusion by eliminating fusion preventing dependences [2, 21]. Fraboulet et al. and Song et al. use loop alignment for minimizing storage [9, 18]. Fraboulet et al.'s algorithm assumes perfectly nested loops and it applies to one loop level only. They do not present experimental results that show the impact of their memory reduction on cache performance and execution time. Both Fraboulet et al. and Song et al. used polynomial-time solvable network flow algorithms. We use a simpler but effective way to reduce storage. Preliminary experimental results show that our alignment strategy is effective not only for fused codes, but also reduces the memory requirements and significantly improves the overall performance of codes that are blocked, unroll-and-jammed, or time skewed as well.

The idea of reducing storage to improve memory performance is not new; the literature already contains numerous papers about this topic [10, 13, 18, 14, 22, 17]. Lewis et al. [13], proposed to use 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 [10]. They first partitioned a collection of loop nests into fusible clusters using a max-flow min-cut algorithm. Loop fusion and array contraction are performed on all loop nests within a cluster. Their array contraction replaces arrays with scalars. They only considered fusion of conformable loop nests that contain exactly the same set of loops and do not apply loop alignment to remove fusion-preventing dependences.

Lim et al. used affine partitioning to support blocking and array contraction [14, 15]. They contract arrays to scalars or lower-dimensional arrays within each independent thread and expand them by block size when blocking is applied to interleave threads. Our storage reduction framework is more general and it is tightly-integrated with other transformations.

Pike and Hilfinger [17] combined 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 are integrated with a more comprehensive set of transformations.

Song et al. presented a network flow algorithm which provably minimizes the memory requirements for multi-dimensional cases [18]. To achieve that, loops are coalesced together into single-level loops before they are fused and after loop shifting is applied [18]. Loop trip counts are assumed to be equal for the corresponding loops, otherwise loops need to be pre-peeled and partitioned. The disadvantage of using loop coalescing is that it may introduce more subscript computation overhead and if-conditions inside the loop body. Fusion is controlled, in their framework, by a heuristic to avoid register spilling and to avoid increasing cache misses by over-fusion. Our storage reduction does not aim to reduce storage to the minimum possible; instead, we reduce it to near minimum while generating efficient code. Our storage reduction is integrated with fusion, blocking, and unroll-and-jam, which enable us to apply fusion more aggressively with less worry about register spilling and cache replacement. Song et al. considered storage reduction only in conjunction with fusion alone.

Previously, live variable analysis has been used for dead code elimination and register allocation [12, 6, 1]. In this work, we use live range analysis to identify temporary variables that can share storage because they are not simultaneously live.

To represent data and iteration sets, our tool makes extensive use of the Omega library [11]. Omega enables us to perform complex iteration space transformations with relative ease.

3 Storage Reduction

In scientific programs that perform complex mathematical calculations, temporary[1] arrays are commonly used to hold intermediate results that will be needed later. However, a large memory footprint for temporary arrays can severely hurt application performance by reducing cache and TLB reuse and thereby increasing a program's memory bandwidth requirements. Here, we describe strategies for improving program efficiency by reducing the memory hierarchy footprint of temporary arrays.

We use a pair of automatic and synergistic optimizations for this purpose. After loop fusion creates opportunities for array contraction, we contract the number and extent of array dimensions to reduce the storage of individual temporary arrays. The goal of this optimization is to improve performance, not to minimize storage; thus, our contraction transformation is careful to ensure that we can index reduced storage in a simple and efficient way. Next, we apply a storage sharing optimization that overlays storage for temporary arrays that are not simultaneously "live". In the rest of this section, we describe these strategies in more detail.

3.1 Effective Array Contraction

Papers in the literature describe how to combine array contraction with loop fusion and blocking. In this section, we describe how we integrate array contraction not only with fusion and blocking, but also with additional loop transformations including unroll-and-jam, time skewing, and iteration space splitting. All of these optimizations are implemented in a source-to-source transformation tool.

Array contraction only applies to temporary arrays in a program. When an array is contractable, if it can't be replaced by a scalar, it can be transformed into an array with fewer dimensions or dimensions of shorter extent. In the rest of this section, we present the rules our source-to-source transformation tool uses to apply array contraction for loops. These principles also apply to acyclic program regions as a special case. Rule 1 identifies uncontractable dimensions. Rules 2–5 describe how to compute a reduced extent for each contractable dimension when contraction is applied in conjunction with various loop transformations.

Rule 1 (Contractability) Array a is not contractable in dimension dim within loop l, where dim is a dimension of a that has index variable of l if 1) a is not a temporary variable in l; 2) dim has multiple index variables; 3) l is used in multiple dimensions of a; 4) loop splitting is applied to a loop enclosing l; 5) there is a flow dependence δ on a. $dist_l(\delta)$ [2] is not constant; 6) l' is an outer loop of l. $d(i) = |max\{dist_i(\delta) \mid \delta$ is a flow dependence on a in $i\}|$, $d(l') > 0$. l' is blocked [3] or l is not blocked; 7) l' is an inner loop of l. $d(i) = |max\{dist_i(\delta)$ $\mid \delta$ is a flow dependence on a in $i\}|$, $d(l') > 0$. l' is blocked, but l is not blocked.

Rule 2 (Direct, Fusion) Assume array a is contractable in dim within loop l. l is an original loop or one

[1] We use the term *temporary* to mean a program variable that is defined before being used within a program region and is not accessed outside that region.

[2] $dist(\delta)$ is distance vector of dependence δ and $dist_l$ is the distance component corresponding to loop l. For each dependence δ, $A(f_{src}(\vec{i}))\ \delta\ A(f_{sink}(\vec{j}))$, where f_{src} and f_{sink} are subscript expressions of the two array references, dependence distance $dist(\delta)$ is defined as $f^{-1}_{sink}(f_{src}(\vec{i}))$ - $\vec{i}$.

[3] When there are multiple loops blocked in a loop nest, we place the outer loops over blocks in the same order as their original ones. Other orders of these loops can be implemented by appling loop permutation to them when it is legal to do so.

```
do j = 1, n
  do i = 1, n
    b(i,j) = a(i,j)
do j = 1, n
  do i = 2, n-1
    c(i,j) = b(i-1,j) + b(i+1,j)
```

a)

```
do j = 1, n
  do i = 1, n
    b(i) = a(i,j)
  do i = 2, n-1
    c(i,j) = b(i-1) + b(i+1)
```

b)

```
do j = 1, n
  do i = 1, n
    b(iand(i,3)) = a(i,j)
    if (i .ge. 3) then
      c(i-1,j) = b(iand(i-2,3)) + b(iand(i,3))
```

c)

```
do j = 1, n
  do i = 1, 2
    b(iand(i,3)) = a(i,j)
  do i = 3, n
    b(iand(i,3)) = a(i,j)
    c(i-1,j) = b(iand(i-2,3)) + b(iand(i,3))
```

d)

Figure 1: Array contraction of fused code.

resulting from fusion. *dim* is the dimension where index variable of l is used. $d(i) = |max\{dist_i(\delta) \mid \delta$ is a flow dependence on a within loop $i\}|$. Then the reduced extent of dimension *dim* is s, $s \geq d(l) + 1$;

Rule 3 (Blocking) Assume array a is contractable in *dim* within loop l. *dim* is the dimension where index variable of l is used. $d(i) = |max\{dist_i(\delta) \mid \delta$ is a flow dependence on a in loop $i\}|$. l' and l'' are an outer and inner loop of l. Then reduced extent of dimension *dim* is s, $s \geq blk_l + d(l)$ if $d(l') > 0$, l is blocked, but l' is not blocked, or if $d(l'') > 0$ and both l and l'' are blocked, where blk_l is the block size of l.

Rule 4 (Unroll-and-Jam) Assume array a is contractable in *dim* within loop l. *dim* is the dimension where index variable of l is used. $d(i) = |max\{dist_i(\delta) \mid \delta$ is a flow dependence on a in loop $i\}|$. l'' is an inner loop of l. If $d(l'') > 0$ and l is unroll-and-jammed, then the reduced extent of dimension *dim* is s, $s \geq uf_l + d(l)$, where uf_l is the unroll factor of l.

Rule 5 (Time Skewing) Assume array a is contractable in *dim* within loop l. *dim* is the dimension where index variable of l is used. $d(i) = |max\{dist_i(\delta) \mid \delta$ is a flow dependence on a within outer time loop $t\}|$ if i is *time skewable* or $d(i) = |max\{dist_i(\delta) \mid \delta$ is a flow dependence on a within $i\}|$ otherwise. l' and l'' are an outer and inner loop of l. Then the reduced extent of dimension *dim* is s, $s \geq (ut - lt) \times ts_l + blk_l + d(l)$ if $d(l') > 0$, l is blocked, but l' is not blocked, or if $d(l'') > 0$ and both l and l'' are blocked, where blk_l and ts_l are the blocking size and time skewing factor of l.

Our rules show how we integrate array contraction with fusion, blocking, unroll-and-jam, and time skewing. Fusion enables array contraction. Blocking, time skewing, and unroll-and-jam may help or limit contraction. Based on the rules described above, we have the following algorithm to contract arrays in a loop.

```
ArrayContraction(l: loop)
1. Identify uncontractable arrays in l based on Rule 1;
2. Set the extent of each contractable dimension of arrays in l based on Rule 2-5;
3. Generate code with contracted arrays or scalars.
```

An example of how array contraction can be applied to fused loops is shown in Figure 1. Figure 1(a) is the original code with two loop nests. Loops j at outer loop level can be fused because there are no flow- and anti-dependences carried by j. Assume array b is not live after the code. Array b is contractable in the second dimension indexed by loop j. The code after applying array contraction to the fused code is shown in Figure 1(b). Furthermore, we can fuse the inner loops i as well by aligning loop i in the second loop nest with an alignment factor of 1. As a result, the first dimension of b can also be contracted in the fused i loop. Three elements of

```
do n = 1, nsmall
 do j = 1, ny - 1
  do k = 1, nz - 1
   do i = 1, nx - 1
     div(i,k,j) = ...
     ft(i,k,j) = ... - div(i,k,j)
 ...
 do j = 1, ny - 1
  do k = 2, nz - 1
   do i = 1, nx - 1
     w(i,k,j) = ...
        + eps0 * (ft(i,k,j) - ft(i,k-1,j))
 ...
 do j = 1, ny - 1
  do k = 1, nz - 1
   do i = 1, nx - 1
     v(i,k,j) = ... - kdivy(j)
        * (div(i,k,j) - div(i,k,j-1))
```

a)

```
do jj$ = 1, ny + 15, 16
 do n = 1, nsmall
  do j = -n+jj$+1, min(ny-1, -n+jj$+16)
   ...
   do k = 1, nz - 1
    do i = 1, nx - 1
      div$(i,k,mod(j,nsmall+16)) = ...
      ft$(i,k) =
         ... - div$(i,k,mod(j,nsmall+16))
   ...
   do k = 2, nz - 1
    do i = 1, nx - 1
      w(i,k,j) =
         ... + eps0 * (ft$(i,k) - ft$(i,k-1)) + ...
   ...
   do k = 1, nz - 1
    do i = 1, nx - 1
      v(i,k,j) = ... - kdivy(j)
         * (div$(i,k,mod(j,nsmall+16))
         - div$(i,k,mod(j-1,nsmall+16)))
```

b)

Figure 2: Array contraction of time skewed code.

array *b* should suffice. We used four elements in the generated code shown in Figure 1(c) because we index the reduced dimension using an *iand* operation, which requires that the length of the reduced dimension be a power of two. Finally, after hoisting the *if* condition out of the innermost level, we get the version shown in Figure 1(d).

The example in Figure 2 shows an application of array contraction to a time skewed code. In Figure 2(a), loop *n* is a time step loop. Inner loops *j* are time skewable. Arrays *div* and *ft* are contractable arrays inside the time loop. After applying time skewing and array contraction, the generated code is shown in Figure 2(b). Array *div* is contracted in the third dimension. Array *ft* is contracted to a 2d array. Because of symbolic term *nsmall*, a *mod* operation has been used for indexing in the code after contraction. we discuss the indexing in the next subsection.

3.2 Indexing Schemes

Generating efficient codes for storage optimizations is important. When our tool reduces the extent of a dimension of a temporary array, it needs to wrap indexing in the reduced dimension. When 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 the instruction count for wrapped indexing. Nevertheless, this strategy increases the pressure on integer registers and can cause register spills. We implemented two indexing schemes in our tool, using *mod* and *iand* operations. Typically, *mod* and *iand* are implemented as intrinsic functions. *mod*(a, b) divides a by b and returns the reminder c with $|c| < |b|$. *iand*(a, b) performs a logical AND of a and b bit by bit. Without doubt, *iand* is less expensive than *mod*. However, using *mod* operation can precisely express the wrapping of the reduced arrays if the dividend could only be either positive or negative. As mentioned earlier, to use *iand*, the reduced extents have to be extended to a power of 2. The significance of the extension depends on the dependence distance and the transformations applied. When the reduced size is not a constant, a conservative upper bound has to be used instead. If an upper bound is not available, the dimension cannot be contracted. In Section 5, we present our preliminary experimental results on performance impact of these two indexing schemes.

In addition to using *iand* and *mod*, our tool can also eliminate all arithmetic overhead for wrapped indexing by replacing subscripts in the reduced dimension with constants in the unrolled code of the core computation. when it is beneficial to do so. In that case, the unroll factor needs to be proportional to the length of the reduced dimension.

```
do k = 1,nz-1
  do j = 1,ny-1
    do i = 1,nx-1
       a(i,j,k) = ....
       ...
do k = 1,nz-1
  do j = 1,ny-1
    do i = 1,nx-1
       b(i,j,k) = ....
       ...
```

(a) Original code

```
equivalence(a,b)
do k = 1,nz-1
  do j = 1,ny-1
    do i = 1,nx-1
       a(i,j,k) = ....
       ...
do k = 1,nz-1
  do j = 1,ny-1
    do i =1,nx-1
       b(i,j,k) = ....
       ...
```

(b) Sharing space between constant-size arrays

```
do k = 1,nz-1
  do j = 1,ny-1
    do i = 1,nx-1
       a(i,j,k) = ....
       ...
do k = 1,nz-1
  do j = 1,ny-1
    do i = 1,nx-1
       a(i,j,k) = ....
       ...
```

(c) Sharing space between adjustable-size arrays

Figure 3: Live range based storage reduction.

3.3 Storage Sharing

In this section, we describe another storage optimization, *storage sharing*, to reduce memory footprint of temporary arrays. To identify the temporaries that can share the same storage with each other, we use live range analysis to compute the range where a variable is live. In general, two values v and $v^{'}$ can share the same storage if their live ranges do not overlap with each other. The following algorithm briefly describes creation of groups of short-lived and storage sharable variables in a program region. We denote live range of variable a as [$Range^b(a)$, $Range^e(a)$] [4].

```
**ComputeArrayLiveRange**($R$: program region)
for each reference $ref$ of local variable $a$ in $R$
    $cfgId$ is the cfg node of the statement including $ref$
    if ($ref$ is the first occurrence of $a$)
        $[Range^b(a), Range^e(a)] = [cfgId, cfgId]$;
    else $[Range^b(a), Range^e(a)] = [predom\_lca(cfgId, Range^b(a)), posdom\_lca(cfgId, Range^e(a))]$;
```

```
**CreateSharableClassSet**($R$: program region)
1. initial sharable class set: $CS = \emptyset$;
2. for each reference $ref$ of local variable $a$ in $R$
    for each class $C \in CS$
        if ($a$ can share storage with any variable $v$ in $C$)
            add $a$ into $C$;
        else create a new class $C' = \{a\}$;
```

Figure 3 shows an example of application of storage sharing. In Figure 3(a), the original code has two loop nests. There are two temporary arrays a and b. Array a is computed and used only in the first loop nest, array b is computed and used only in the second loop nest. Both are not alive outside the loops enclosing their definitions and uses. By live range analysis, we find a's live range does not overlap with b's live range; thus, they can share storage with one another. Depending on how they are defined, the original code can be transformed into the code in Figure 3(b) by using an equivalence statement if both a and b are not adjustable sized arrays or otherwise into the code in Figure 3(c) where every instance of b has been replaced by an instance of a. (This is possible in this example because the size of a and b are the same.)

Although forcing the two values to share the storage may increase register pressure for scalars, it may substantially reduce size of data footprint of a program if they are stored in arrays.

[4]Here, "b" denotes "begin" and "e" denotes "end".

```
[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 4: An example of statement motion.

4 Enabling Transformations

In this section, we describe two enabling transformations for the storage optimizations. Statement motion enables fusion of loops by moving away statements between them. A larger fused loop often exposes more temporaries for storage optimizations. Necessary and sufficient alignment is a strategy that we use to enable multi-level fusion of neighboring loops and more aggressive array contraction.

4.1 Statement Motion

In real scientific applications, loops can be arbitrarily nested and statements may exist inside or outside a loop, or between a sequence of loop nests. If there are statements between two loops, these two loops may not be fused in a straightforward way. Although we can always embed the statements between the loops into one of the loops, fused loop may not be efficient enough when if-conditions have to be inserted inside the loop body. We developed an optimization called *statement motion*, based on program's data and control flows, to move forward statements between loops to enable fusion.

```
StmtMotion($P$: program)
1. for each statement $s$ in $P$
    if ($s$ is an assignment between loops)
       FindDestination($s$, map);
2. for each statement $s$ in the program
    if ($s$ is in map and map($s$) is not $s$)
       create a temporal variable var for the definition def in $s$;
       replace def and each use that def reaches by var;
       find final destination dest of s through map if map($s$) needs move;
       MoveStmtToDest($s$, dest);

FindDestination($s$: statement, map: motion map)
1. if ($s$ is not a scalar assignment or
      the definition def in $s$ is not the only reaching definition of its use)
      set map($s$) to $s$;
2. find the set of reaching definitions, rDefs, of all the uses in $s$;
3. find the post-dominator pdom of the def nodes in rDefs;
4. if (pdom pre-dominates $s$)
      set map($s$) to pdom;
   else FindDestInStmtList($s$, map, dest);
```

StmtMotion first finds the destination for each statement that may need to move and builds a motion map. Then it moves each of the statements to its destination based on the motion map. **FindDestination** first checks the definition of a statement. We only move scalar assignments in our current implementation. If the def is not the only reaching definition that can reach its uses, we will not move the statement. Then, it checks the uses of the statement and finds the post-dominator of the reaching definitions of the uses in the statement. If the post-dominator predominates the statement, we set the post-dominator as the destination of the moving statement. Otherwise, it checks the statements before the moving statement in reverse order and sets a statement as the destination if the statement contains a reaching definition of a use in the moving statement or has an exit out of the current level of statement list. The example in Figure 4 shows how statements `s1`, `s2`, and `s3` are moved and code is transformed. After statement motion, both the i loops and j loops can be aligned and fused.

4.2 Necessary and Sufficient Alignment

In this section, we describe a new loop alignment strategy called *necessary and sufficient alignment* that we have 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. A brief description of the algorithm is as follows.

```
NecessarySufficientAlignment(lList: a list of loops to be fused)
1. SufficientAlignment(lList);
2. NecessaryAlignment(lList);

NecessaryAlignment(lList)
for each loop $i$ in lList in reverse order
    $backalignFactor(i) = 0$;
    if applying necessary alignment to $i$ is beneficial
        for each succeeding loop $j$ of loop $i$
            $depDist = min\{dist_i(\delta) \mid \delta$ is a dependence on an array $a$ from $i$ to $j\}$;
            $backalignFactor(i) = min(backalignFactor(i),\ depDist - (alignFactor(i) - alignFactor(j)))$;
```

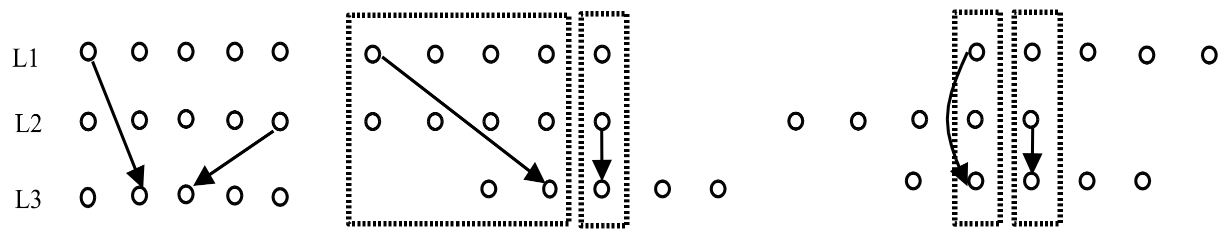
The **NecessaryAlignment** is only applied when it is beneficial by which we mean it may reduce storage for temporary arrays. The initialization $backalignFactor(i) = 0$ is to ensure the legality of the transformation. $dist(\delta)$ is distance vector of dependence δ as described in Section 3.1. *alignFactor* is a sufficient distance vector.

Figure 5 shows the applications of a sufficient alignment only and a necessary and sufficient alignment. In Figure 5(a), there are three neighboring loops L1, L2, and L3. Cycles represent iterations and arrows represent dependences between the loops. Because of the fusion preventing dependence from L2 to L3, these two loops cannot be fused. By applying a sufficient alignment to L3, all three loops can be fused together. After fusion, the distances of the dependences from L1 and L2 are 3 and 0 as shown in Figure 5(b). Assume array a and b are temporaries and the dependences are the only ones carried on them. Then, the size of a and b can be reduced to four and one elements as the grey dot boxes show. On the other hand, if we apply a necessary and sufficient alignment to shift the first loop three iterations right as well. The result after alignment is shown in Figure 5(c). Since the distances of both dependences are 0 now, one element for each of a and b should suffice. Although the effect of array contraction is not significant in this case, the purpose of the example is to show the main idea. When the temporary arrays are high dimensional or the array contraction is applied with other transformations (blocking or unroll-and-jam), the reduction could be substantial.

Necessary and sufficient alignment enable array contraction of codes that are fused, blocked, time skewed. It is also beneficial for guard-free core generation by enlarging the core and reducing the number of clipped tiles.

5 Experimental Results

To evaluate the effectiveness of the storage reduction optimizations, we present experimental results of three programs: a Runga-Kutta advection kernel `twostages` for mesoscale weather modeling, the Livermore Loop 18 benchmark (`LL18`), an explicit hydrodynamics kernel, and a key routine `smlstep` of a weather application code NCOMMAS that was created by the National Severe Storms Laboratory [19] to study cloud dynamics



```
[L1]  do i = 1, n
        a(i) = ...
[L2]  do i = 1, n
        b(i) = ...
[L3]  do i = 1, n
        c(i) = a(i-1)+b(i+2)
```

(a) Before alignment

```
do i = 1, n
  a(i) = ...
do i = 1, n
  b(i) = ...
do i = 3, n+2
  c(i-2) = a(i-3)+b(i)
```

(b) After sufficient alignment

```
do i = 4, n+3
  a(i-3) = ...
do i = 1, n
  b(i) = ...
do i = 3, n+2
  c(i-2) = a(i-3)+b(i)
```

(c) After necessary and sufficient alignment

Figure 5: Necessary and sufficient alignment.

on a variety of scales. `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. `smlstep` is one of the key routine of the NCOMMAS code. It has 260 lines of Fortran code with 30 loops nested at deepest level of 4. The routine uses 9 three-dimensional single precision arrays, among them two are temporaries. It includes a time step loop with variable step sizes and a loop sweeping along alternating directions. 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 the NCOMMAS code are $51 \times 51 \times 36$ and $81 \times 81 \times 36$.

Experiments were conducted on multiple versions of each tested code, both original or optimized generated by our tool. In the following discussion and tables shown, we use the notation *n*`lF` to denote versions with n-level fusion applied. In addition, `BL, UJ, AL, AC, SR, TS` represents application of blocking, unroll-and-jam, necessary and sufficient alignment, array contraction, storage reduction, or prismatic time skewing separately. We use "`+`" to represent a combination of multiple optimizations. In our experiments, fusion was applied to all optimized versions for each tested code. So was the statement motion as an enabling transformation for more aggressive loop fusion. The necessary and sufficient alignment was only used for storage optimizations.

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. 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 and a graphical memory usage viewer *gmemusage*. *gmemusage* displays overall memory usage as well as a breakdown to text, heap and stack spaces of a running executable. 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. Additional driver programs were added to both `twostages` and `LL18` for testing. Two iterations of `twostages` and five iterations of `LL18` were executed from the drivers. We run three iterations of the entire solver of the NCOMMAS. The variable time step size is 2, 3, or 6.

5.1 Overall Performance

We tested overall performance gain from the storage optimizations by comparing cache and TLB misses, memory use, CPU cycles of versions with and without storage optimized. Tables 1–3 show results of incrementally applying fusion, blocking, unroll-and-jam, time skewing, and storage reduction to the three test codes. Time skewing was only applied in `smlstep` since it is the only test code with a time step loop. Blocking and unroll-and-jam

Code	L1 misses(M)		L2 misses(M)		TLB misses(K)		Memory(MB)	Cycles(M)		Speedup	
	`-SR`	`+SR`	`-SR`	`+SR`	`-SR`	`+SR`	`+SR`	`-SR`	`+SR`	`+SR/-SR`	`+SR/Orig`
64 × 64 × 64											
`Orig`	3.10		1.38		132		25.5	256			
`1lF`	3.89	4.41	0.88	0.48	71.2	19.3	12.2	184	124	1.48	2.06
`2lF`	5.16	3.23	0.77	0.45	77.4	26.7	12.2	163	119	1.37	2.15
`2lF+UJ`	3.90	3.85	0.94	0.67	43.4	99.9	14.3	194	143	1.36	1.79
`2lF+BL`	5.21	3.34	0.63	0.49	89.7	37.8	12.6	142	123	1.15	2.08
`2lF+UJ+BL`	3.94	3.94	0.90	0.52	61.2	112	14.3	189	123	1.54	2.08
128 × 128 × 128											
`Orig`	20.0		8.43		578		150	1598			
`1lF`	25.9	27.7	9.25	8.70	325	138	71.2	1650	1591	1.04	1.00
`2lF`	35.4	22.4	9.32	7.61	658	190	70.4	1677	1437	1.17	1.11
`2lF+UJ`	24.0	25.8	8.42	5.28	244	513	82.1	1557	1039	1.50	1.54
`2lF+BL`	35.5	23.2	4.75	3.11	779	244	72.1	1004	781	1.29	2.05
`2lF+UJ+BL`	24.8	26.0	5.22	3.56	338	550	82.1	1115	800	1.39	2.00

Table 1: Performance comparison of optimized versions over the original version of `twostages`.

were not applied to `smlstep` because of the loops sweeping along alternating directions at an inner loop level. Memory usage was not presented in Table 3 since we cannot isolate the execution of `smlstep` from the rest of the NCOMMAS code. In each table, the leftmost column describes the transformations used for different test cases. Columns marked "`-SR`" and "`+SR`" shows results for version with and without storage reduction applied additionally. Memory usage of transformed codes without applying storage reduction is same as that of the original code. Results of cache misses, CPU cycles, and TLB misses are presented in million(M), or thousand(K), while memory usage is presented in megabytes(MB). The speedup columns in each table shows execution time speedups that compare the performance of the transformed codes with and without storage reduction and the performance of the transformed code with storage reduction and the original code.

As the tables show, storage reduction optimizations integrated very well with the rest of the transformations. Although fusion, blocking, unroll-and-jam, and time skewing, as individual optimizations or applied in combination, significantly improved the performance of the original test codes in most cases, the storage optimizations speedup performance in all cases. By reducing up to half of the data footprint, it substantially reduces both L2 cache misses and TLB misses. As a result, it achieves additional performance improvement up to a factor of 2, leading the overall speedups over the original codes up to a factor of 2.62. It is noteworthy that storage reduction reduced TLB misses of an unroll-and-jammed code of `LL18` at the large data size by a factor of 9.8. This is due to significant reduction of huge number of conflict TLB misses of the unroll-and-jammed code. On other hand, it increases TLB misses from 63% up to a factor of 2.3 in the four test cases of `twostages` applied unroll-and-jam. This is due to an increase of conflict TLB misses after applying storage sharing. However, storage optimizations speed up overall performance of unroll-and-jammed codes by 14%-50%. Small improvement for the larger problem size of `twostages` indicates that the reduced data size is still relatively large to data reuse across an outer loop. The relative small improvement over time skewed versions are due to already highly improved data reuses in the time skewed versions. Since unroll-and-jam is applied to a guard free core in our generated codes and loop splitting is applied to clip the tiles along boundaries, storage reduction was not able to reduce size of one of the array `athird`, causing over 10% more of memory usage compared to other storage reduced versions.

5.2 Evaluation of Necessary and Sufficient Alignment

The overall performance gains of storage reduction shown in Table 1– 3 were achieved through a combination of optimizations. Among them is the necessary and sufficient alignment. To evaluate its impact on performance, we compared performance results of different transformed versions with and without applying the necessary and sufficient alignment. Table 4 shows the results of `twostages`. The first column shows the transformations used for different test cases. Array contraction was applied in each version of the transformed codes. However, no live range based storage sharing was applied. We used `iand` for indexing all contracted dimensions. For each metric

Code	L1 misses(M)		L2 misses(M)		TLB misses(K)		Memory(MB)	Cycles(M)		Speedup	
	`-SR`	`+SR`	`-SR`	`+SR`	`-SR`	`+SR`	`+SR`	`-SR`	`+SR`	`+SR/-SR`	`+SR/Orig`
512 × 512											
`Orig`	8.22		2.17		25.2		18.7	423			
`1lF`	8.79	8.26	1.34	0.82	23.0	10.9	10.3	297	218	1.36	1.94
`2lF`	9.17	7.16	1.35	0.82	22.8	10.3	10.3	345	248	1.39	1.71
`2lF+UJ`	7.57	5.73	1.36	0.87	43.9	19.0	10.4	295	239	1.23	1.77
`2lF+BL`	7.66	5.49	1.42	1.04	476	375	10.7	326	244	1.34	1.73
`2lF+UJ+BL`	4.52	4.70	2.04	1.04	680	488	10.7	414	313	1.32	1.35
1K × 1K											
`Orig`	39.5		8.62		126		74.2	1767			
`1lF`	40.7	37.7	5.47	3.65	145	67.0	41.1	1286	952	1.35	1.86
`2lF`	38.6	33.1	5.44	3.41	178	65.7	41.1	1518	1074	1.41	1.65
`2lF+UJ`	45.8	45.6	5.86	4.73	5082	519	42.2	1515	1333	1.14	1.33
`2lF+BL`	36.3	24.2	5.56	4.94	5643	4466	41.8	1491	1218	1.22	1.45
`2lF+UJ+BL`	48.3	45.3	8.19	5.03	19020	8717	41.8	2104	1719	1.22	1.03

Table 2: Performance comparison of optimized versions over the original version of `LL18`.

Code	L1 misses(M)		L2 misses(M)		TLB misses(K)		Cycles(M)		Speedup	
	`-SR`	`+SR`	`-SR`	`+SR`	`-SR`	`+SR`	`-SR`	`+SR`	`+SR/-SR`	`+SR/Orig`
51 × 51 × 36										
`Orig`	10.5		4.50		733		832			
`1lF`	10.0	10.1	4.57	1.97	592	107	856	423	2.02	1.97
`2lF`	10.2	10.1	2.27	1.84	112	102	470	401	1.17	2.07
`1lF+TS`	10.2	10.1	1.34	1.05	113	107	351	318	1.10	2.62
81 × 81 × 36										
`Orig`	64.5		11.6		1238		2513			
`1lF`	58.2	77.1	11.3	4.98	1018	220	2491	1494	1.67	1.68
`2lF`	78.9	76.2	5.54	4.56	235	210	1658	1457	1.14	1.72
`1lF+TS`	79.2	76.3	3.69	3.34	239	223	1410	1298	1.09	1.94

Table 3: Performance comparison of optimized versions over the original version of `smlstep`.

shown in the table, the columns marked as "`-AL`" and "`+AL`" represent versions without and with the advanced alignment. The last column shows performance speedups by applying the alignment to different transformed codes.

As Table 4 shows, a transformed code could perform 17% to 58% better if the advanced alignment is used. This is mainly because array contraction becomes more effective when loops are aligned with necessary and sufficient alignment factors. Array contraction without the necessary and sufficient alignment is less effective in cases where dependence distance vectors have multiple nonzero components. In fact, we cannot contract any arrays in unroll-and-jammed versions of `twostages` because unroll-and-jam is only applied to guard free core and tiles including the core and boundary tiles are executed in lexicographical order after applying loop splitting in our generated codes. For other cases, only the third dimension can be contracted and limited contraction has little impact on cache misses when data size is large. Furthermore, reducing size to a power of 2 (32 for blocked code with blocking size 16) potentially introduces additional cache conflict misses.

5.3 Evaluation of Indexing Schemes

So far, we have shown that contracting arrays after aligning loops with necessary and sufficient alignment factors, and indexing contracted dimensions with `iand` achieve significant performance improvement. When we use `iand`, we expand size in each contracted dimension to a power of 2. The extra space introduced depends on many

Code	L1 misses(M)		L2 misses(M)		TLB misses(K)		Cycles(M)		Speedup
	`-AL`	`+AL`	`-AL`	`+AL`	`-AL`	`+AL`	`-AL`	`+AL`	`+AL/-AL`
64 × 64 × 64									
`1lF+AC`	4.37	4.41	0.65	0.54	22.4	20.1	156	130	1.20
`2lF+AC`	4.09	3.30	0.62	0.46	49.6	27.5	151	120	1.26
`2lF+UJ+AC`	4.02	2.96	0.93	0.74	41.6	27.2	186	146	1.27
`2lF+BL+AC`	4.11	3.48	0.75	0.50	77.9	38.3	175	124	1.41
`2lF+UJ+BL+AC`	4.12	3.16	0.87	0.55	61.9	40.9	181	123	1.47
128 × 128 × 128									
`1lF+AC`	27.5	27.7	11.1	9.48	145	141	1998	1708	1.17
`2lF+AC`	29.7	22.6	10.7	7.63	341	190	2000	1450	1.38
`2lF+UJ+AC`	26.6	20.0	8.37	6.22	261	161	1502	1144	1.31
`2lF+BL+AC`	30.1	23.4	5.54	3.13	488	247	1238	785	1.58
`2lF+UJ+BL+AC`	26.8	20.8	5.34	3.88	354	213	1092	820	1.33

Table 4: Performance comparison of optimized versions of `twostages` with and without necessary alignment.

Code	L1 misses(M)		L2 misses(M)		TLB misses(K)		Cycles(M)		Speedup
	`iand`	`+mod`	`iand`	`+mod`	`iand`	`+mod`	`iand`	`+mod`	`+mod/iand`
64 × 64 × 64									
`1lF+AC`	4.41	4.41	0.54	0.53	20.1	20.3	130	130	1.00
`2lF+AC`	3.30	3.27	0.46	0.46	27.5	26.5	120	119	1.01
`2lF+UJ+AC`	2.96	3.11	0.74	0.72	27.2	27.2	146	143	1.02
`2lF+BL+AC`	3.48	3.38	0.50	0.50	38.3	38.3	124	126	0.98
`2lF+UJ+BL+AC`	3.16	3.31	0.55	0.54	40.9	40.1	123	121	1.02
128 × 128 × 128									
`1lF+AC`	27.7	27.7	9.48	9.55	141	141	1708	1721	0.99
`2lF+AC`	22.6	22.9	7.63	7.61	190	193	1450	1453	1.00
`2lF+UJ+AC`	20.0	20.4	6.22	6.24	161	161	1144	1161	0.99
`2lF+BL+AC`	23.4	23.8	3.13	3.10	247	245	785	782	1.00
`2lF+UJ+BL+AC`	20.8	21.0	3.88	3.55	213	226	820	773	1.06

Table 5: Performance comparison of optimized versions of `twostages` with iand and mod.

factors including dependence distances, blocking size, unroll factor, and others. Conservative upper bounds have to be used when the reduced extents include symbolic terms. To avoid the additional space, our tool can also generate codes using `mod` indexing schemes. Different from `iand`, `mod` is more flexible. But its implementation is more expensive. In cases where subscripts in the contracted dimensions could be of both positive and negative values, we need to add offsets to the subscripts before the `mod` operation. To evaluate performance impact of extra space and indexing efficiency, we experimented with `iand` and `mod` on multiple versions of `twostages` and performance comparison of the results is presented in Table 5. The columns marked as `iand` represent versions using `iand` for all contracted dimensions, whereas the columns marked as `mod` represent versions that use `mod` for the dimensions which cannot be tightly contracted with `iand`.

From Table 5, we found that using `mod` to replace `iand` whenever more aggressive array contraction is possible does not improve overall performance much in all case except for the blocked and unroll-and-jammed version which performs 6% better with `mod` for the larger data size. This indicates that extra space introduced with `iand` indexing scheme is relatively small compared to the overall contracted space although it could be significant to each individual array. Second, moderate use of `mod` operation won't cause much overhead, particularly if they can be hoisted to outer loop levels by underlying compilers.

Code	L1 misses(M)		L2 misses(M)		TLB misses(K)		Cycles(M)		Speedup
	`-LR`	`+LR`	`-LR`	`+LR`	`-LR`	`+LR`	`-LR`	`+LR`	`+LR/-LR`
64 × 64 × 64									
`1lF+AC`	4.41	4.42	0.53	0.48	20.3	19.3	130	120	1.08
`2lF+AC`	3.27	3.23	0.46	0.45	26.5	26.7	119	119	1.00
`2lF+UJ+AC`	3.11	3.85	0.72	0.67	27.2	99.9	143	143	1.00
`2lF+BL+AC`	3.38	3.30	0.50	0.49	38.3	39.1	126	124	1.02
`2lF+UJ+BL+AC`	3.31	3.94	0.54	0.52	40.1	118	121	123	0.98
128 × 128 × 128									
`1lF+AC`	27.7	27.7	9.55	8.70	141	138	1721	1591	1.08
`2lF+AC`	22.9	22.4	7.61	7.61	193	190	1453	1437	1.01
`2lF+UJ+AC`	20.4	25.7	6.24	5.24	161	529	1161	1046	1.11
`2lF+BL+AC`	23.8	23.2	3.10	3.11	245	244	782	781	1.00
`2lF+UJ+BL+AC`	21.0	26.0	3.55	3.56	226	550	773	800	0.97

Table 6: Performance comparison of optimized versions of `twostages` with and without live range based storage sharing.

5.4 Evaluation of Live Range Based Storage Sharing

The live range based storage sharing was only applied to `twostages` in our experiments. By computing live range of each temporary array variable in the code, we are able to identify two groups of two dimensional variables in the one level fused code and one group of one dimensional variables in the two level fused versions that can potentially share their storages. Since they are adjustable sized arrays, variables in the same group are replaced by a single variable. Table 6 shows a performance comparison between versions with and without storage sharing applied. Array contraction has been applied to all tested versions as a base optimization. `iand` is used if the reduced extent of contracted dimension is a power of 2, `mod` is used otherwise. Columns marked as "`+LR`" and "`-LR`" represent results of transformed versions with and without storage sharing applied. Overall, storage sharing improves performance by up to 11%. It may also slow down execution in some cases where sharing storage causes substantial conflict cache and TLB misses.

6 Conclusions and Future Work

We have described an integrated framework for storage optimizations to reducing data footprint. By using two enabling transformations, namely statement motion and necessary and sufficient alignment, we are able to aggressively contract temporary arrays and share storage for those temporaries whose live ranges do not overlap. We also developed and investigated two indexing schemes for addressing contracted array dimensions. The storage reduction optimizations integrate well with loop transformations performed by our source-to-source transformation tool. Experimental results show that with our new storage optimizations, we achieved much better overall performance than that achievable by only applying any individual or combination of the other loop reordering transformations.

Although our storage reduction strategy can be useful for optimizing existing applications, a key benefit of our techniques is that they can simplify the task of developing new application code. To cope with the complexity of writing scientific applications, application developers would often prefer to factor complex calculations into a series of small steps with each step performing a simple computation to advance some aspect of the program state. Such a coding style leads to programs that are easy to understand and maintain. However, without applying storage reduction techniques such as those we describe, this coding style leads to inefficiency because of the increased memory footprint caused by using temporary arrays—when data sets are large, such codes 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. Using our storage reduction techniques however can enable programmers to write maintainable programs in this preferred factored style and map them into a form that can execute efficiently.

Acknowledgements

This research was supported in part by NCSA under National Science Foundation cooperative agreement ACI-9619019 and the Los Alamos National Laboratory Computer Science Institute (LACSI) through LANL contract number 03891-99-23 as part of the prime contract (W-7405-ENG-36) between the DOE and the Regents of the University of California.

References

[1] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, second edition, 1986.

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

[3] 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.

[4] 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.

[5] S. Coleman and K. S. McKinley. Tile size selection using cache organization. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[6] K. Cooper and L. Torczon. *Engineering a Compiler.* Morgan Kaufmann, 2003.

[7] 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.

[8] 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.).

[9] A. Fraboulet, G. Huard, and A. Mignotte. Loop alignment for memory accesses optimization. In *Proceedings of the Twelfth International Symposium on System Synthesis*, Boca Raton, FL, July 1999.

[10] 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.

[11] 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.

[12] K. Kennedy. A survey of data flow analysis techniques. In S. Muchnick and N. Jones, editors, *Program Flow Analysis*, pages 5–54. Prentice-Hall, 1981.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.

[21] M. J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, Redwood City, CA, 1996.

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