Increasing Temporal Locality with Skewing and Recursive Blocking

Guohua Jin
Dept. of Computer Science
Rice University
Houston, TX
jin@cs.rice.edu

John Mellor-Crummey
Dept. of Computer Science
Rice University
Houston, TX
johnmc@cs.rice.edu

Robert Fowler
Dept. of Computer Science
Rice University
Houston, TX
rjf@cs.rice.edu

ABSTRACT

We present a strategy, called recursive prismatic time skewing, that increase temporal reuse at all memory hierarchy levels, thus improving the performance of scientific codes that use iterative methods. Prismatic time skewing partitions iteration space of multiple loops into skewed prisms with both spatial and temporal (or convergence) dimensions. Novel aspects of this work include: multi-dimensional loop skewing; handling carried data dependences in the skewed loops without additional storage; bi-directional skewing to accommodate periodic boundary conditions; and an analysis and transformation strategy that works inter-procedurally. We combine prismatic skewing with a recursive blocking strategy to boost reuse at all levels in a memory hierarchy. A preliminary evaluation of these techniques shows significant performance improvements compared both to original codes and to methods described previously in the literature. With an inter-procedural application of our techniques, we were able to reduce total primary cache misses of a large application code by 27% and secondary cache misses by 119%.

1. INTRODUCTION

For a program to achieve high performance on modern computer systems with multi-level memory hierarchies, it must exhibit effective data reuse at each level. Good spatial locality of memory accesses means that most or all data fetched into cache will be used. It is well known that careful data layout can improve spatial locality and mitigate interference. However, on systems with limited memory bandwidth, good spatial locality alone is not enough for high performance.

To achieve high performance on recent systems with typical memory hierarchies, programs must also exhibit substantial temporal locality. That is, on average, each data item will be accessed many times in cache before being evicted.

Tiling loops over a program's data domain is a strategy commonly used to increase both spatial and temporal reuse in one or more levels of cache [8, 10, 13, 15, 16]. Tiling reshapes an iteration space[1] over a data domain by partitioning it into pieces that fit comfortably into cache and then completing all computations on each piece before moving to the next. Tiling rearranges the order of computation so that multiple references to a data element occur in inner loops while that element is still resident in cache. Recursive blocking [5, 11, 17, 23] generalizes multi-level tiling by recursively partitioning a computation's iteration space, thus providing locality at each level of the recursion. This strategy is attractive because it can achieve good performance across a wide range of systems, independent of specific memory hierarchy implementations.

However, even if tiling and recursively blocking are successful at extracting maximal data reuse within each iteration, that is in the loops over the data domain (*spatial loops* or *spatial loop nest*), this is often not sufficient to reduce the program's memory bandwidth requirements enough to attain acceptable performance. For example, if four data elements fit in a cache line and each element is reused five times in the spatial loop nest, the best possible cache miss rate is still five percent for this data. Further increasing the number of references per miss requires either a change of algorithm or exploiting additional data reuses outside the spatial loop nest.

In this paper, we focus on techniques that improve temporal locality in scientific applications that iterate over a regular discretized domain. This type of computation is commonly used for solving partial differential equations and in image processing. The spatial problem domain is discretized by mapping it onto a one, two, or three dimensional array (or arrays). These arrays exceed the size of caches at all levels for large-scale applications. The core computation consists of repeated updates over the spatial domain to each data element based on its current value and the values of some of its neighbors. Around the spatial loops are one or more outer loops that represent time steps in discrete time simulations or convergence steps for indirect solution methods[2]. Instead of allocating data for each time or convergence step, these

[1]Iteration space is the set of all possible iterations for a loop or nested loops

[2]For simplicity, we use the term "time step" to refer to each iteration of such loops, and the term "time step loop" to refer to the loop of time steps.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SC2001 November 2001, Denver (c) 2001 ACM 1-58113-293-X/01/0011 $5.00

```
do t = 1, U_t
   do i_1 = L_{i_1}, U_{i_1}
     ...
       do i_m = L_{i_m}, U_{i_m}
          LOOP BODY_1
   do j_1 = L_{j_1}, U_{j_1}
     ...
       do j_n = L_{j_n}, U_{j_n}
          LOOP BODY_2
```

Figure 1: A skeletal program fragment for an iterative stencil computation.

loops typically reuse the memory locations used by their predecessors. Thus, these are guaranteed to carry many data dependences and their index variables are not used in index expressions in the data domains. Figure 1 shows the skeletal structure of a sample computation of this class consisting of an outer time step loop that encloses two loop nests that sweep over the data domain.

We present a new strategy, called *Recursive Prismatic Time Skewing* (RPTS), which integrates recursive blocking with time skewing to increase temporal reuse in iterative stencil computations. First, we apply a skewing transformation to an iterative stencil computation both to transform loop-carried data dependences into a form that allows recursive blocking and to create opportunities for exploiting temporal reuse across time steps once the computation is blocked. Then we apply recursive blocking to partition the iteration space over the data domain into cache-sized tiles to enhance reuse at all levels of a memory hierarchy.

RPTS has several novel features that distinguish it from other published time skewing techniques. First, it integrates recursive blocking and time skewing. Second, the skewing transformation applies to multiple spatial dimensions. Third, it handles data dependences carried on a "time" loop without additional storage. Fourth, it uses bi-directional skewing for handling periodic boundary conditions. Finally, it is based on an inter-procedural analysis and transformation strategy, thus allowing time step and spatial loops to be defined in different routines.

Experimental results show RPTS improves overall performance by from 12% to 133% compared to alternative strategies from the the literature. Speedups over the baseline applications compiled using standard techniques range from 243% to a factor of 15.

The remainder of the paper is organized as follows. First, we describe background for our research including related work and definitions. Second, we describe our recursive prismatic time skewing transformation. Third, we present our experimental validation of these techniques.

2. BACKGROUND

2.1 Related Work

Compiler techniques for improving memory hierarchy performance have been widely studied. At the register level, scalar replacement [9] can reduce the frequency of loads and stores by enabling back-end compilers to allocate subscripted variables to registers. Combined with loop fusion or other transformations that reshape a program's iteration space by reordering statement instances to bring temporal reuse of values within the same iteration or a few iterations away (e.g. unroll-and-jam [7]), scalar replacement can dramatically reduce load/store traffic.

Tiling (also known as loop blocking) is one of the key transformations used for improving temporal reuse in cache (*e.g.*, [8, 10, 15, 14, 16]). Most research on tiling has focused on improving reuse for a single level of cache and has been only applied to perfectly-nested loops, in which an outer loop encloses exactly one inner loop. To enhance locality in imperfectly-nested loops, Ahmed, Mateev, and Pingali proposed an approach by which iteration space of each statement is embedded in a special space called *product space* using affine embedding functions [3]. These embedding functions effectively generalized transformations like loop fusion and loop fission that has been used for locality enhancement and can be used for tiling imperfectly-nested loops if the product space can be transformed into a fully permutable one [4].

Tiling can also be applied to multi-level memory hierarchies by separately blocking for each level in the memory hierarchy. Kodukula and Pingali [13] describe a data-centric technique for performing multi-level tilings. A key difficulty with multi-level tiling techniques is choosing the appropriate tile sizes. The optimal tile size for a particular level of the memory hierarchy depends on architectural parameters such as cache size and associativity as well as code characteristics such as stencil shape and the number of variables being referenced. Such tilings are best performed by a compiler because having the wrong number of levels of tiling or wrong tile sizes can lead to performance degradation when the tilings are mismatched to the target architecture.

An alternative to multi-level tiling is to use recursive blocking [5, 11, 17]. By recursively partitioning the iteration space to a base tile size guaranteed to be small enough for all systems, this strategy can achieve good performance on a multi-level memory hierarchy regardless of its characteristics, because below some level of the recursion, the size of the data referenced will fit in each level of cache and be available for temporal reuse. Yi, Adve and Kennedy [23] have developed automatic techniques for automatically rewriting loop nests whose data dependences meet certain conditions into a recursive divide-and-conquer form.

For iterative computations, additional temporal reuse can be achieved by interleaving multiple time steps of the computations so that values computed in one time step are reused in the next before they are evicted. This class of techniques has been explored by both Wonnacott [22], and Song and Li [18, 19].

Wonnacott [22] describes *time skewing*, a technique that combines blocking in both the data and time domains. A time step loop is skewed with one spatial loop over a a data tile so as to preserve data dependences across multiple time steps. This technique involves transforming both the iteration space and array indexing functions in a loop body. Key requirements for this method are: (1) all data values consumed in a loop nest must come from the previous iteration of the time step loop; (2) every statement within the time step loop must be nested within the same number of loops, and (3) the flow of values across iterations of the time step loop must be identical for all time steps. The first restriction is the most limiting: Wonnacott's time skewing strategy does not apply when data domain loops carry data dependences.

Song and Li [18, 19] describe a similar technique that uses a combination of loop skewing and tiling, combined with limited array expansion. The goal of this method is to reduce memory traffic by improving temporal data reuse in secondary cache. To achieve this goal, Song and Li choose a blocking factor so that the data accessed by computation over a tile will likely fit in secondary cache. Unlike Wonnacott's time skewing technique, Song and Li's technique is applicable to codes in which spatial loops may carry data dependences.

Prismatic skewing differs from previous time-skewings in three ways. First, prismatic skewing may skew multiple spatial dimensions. Both Wonnacott [22] and Song & Li [19] consider skewing only a single spatial dimension. Second, prismatic skewing handles carried data dependences on spatial dimensions without requiring additional storage. Wonnacott considers only computations that have no carried dependences on spatial dimensions. Song and Li handle carried dependences on the temporal dimension by introducing additional storage using odd-even duplication. Third, prismatic skewing handles periodic boundary conditions using bi-directional skewing. No previous time-skewing work has considered this issue.

In our program model, the time step loop and the spatial loops are not necessary in the same routine. Loops can be imperfectly nested. At each time step, there could be computational sweeps along different directions, data copying, and/or reductions across the data index domain represented by the loop nests enclosed by the time step loop t. Data dependences can be carried by the time step loop or the spatial loops. Without loss of generality, we assume that all loops have stride one.

Prismatic skewing and recursive blocking are also compatible with techniques for enhancing temporal reuse in the registers (e.g., [9]) that can further reduce a program's needs for memory hierarchy bandwidth and improve performance.

2.2 Definitions

A dependence is called a *true dependence* if the source writes a value that is read by the sink, an *anti-dependence* if the source reads a variable that is written later by the sink , or an *output dependence* if both the source and the sink write the same variable. A *dependence distance vector* describes the number of iterations crossed by the dependence for each index of the common enclosing loops. A dependence is *loop-independent* if its distance vector is all "0". Otherwise it is *loop-carried* and is said to be carried by an enclosing loop at the level that corresponds to the first nonzero component in the distance vector.

3. APPROACH

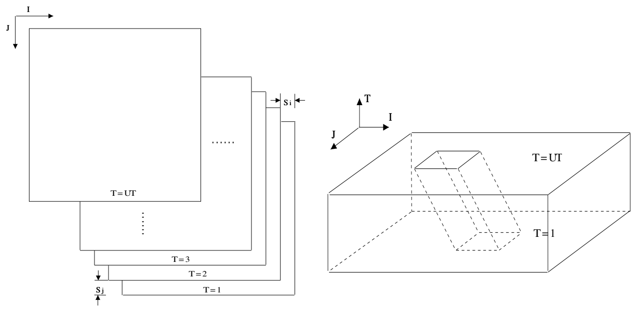

Figure 2: 2D prismatic skewing.

RPTS derives its name from the fact that it skews an iteration space that ranges over time and space and then blocked recursively. The iteration space over the data domain is thus carved into a set of nested, skewed time-space prisms in which one or more spatial dimensions is skewed by the index of a time-step loop. Loops over spatial dimensions may also need to be skewed with respect to one another to enable recursive blocking.

Figure 2 shows a rectangular solid time-space iteration volume and a prismatic tile within which each spatial dimension has been skewed by the temporal dimension. All prismatic tiles of the time-space iteration volume are skewed in the same way. Along the boundaries prismatic tiles are clipped to prevent them from extending outside the iteration space.

In the next subsection, we describe a skewing transformation that has two effects: it skews spatial loops over time to enable reuse across time steps and it transforms data dependences to enable recursive blocking. Following that, we describe our recursive blocking transformation.

3.1 Skewing

The skewing transformation we use serves two purposes. First, it skews spatial loops w.r.t. one another to enable recursive blocking. Second, it skews spatial loops w.r.t. a time step loop which, after blocking the iteration space, will enable temporal reuse of cached values across multiple time steps.

3.1.1 Computing Spatial Skew Factors

We consider two types of skewing for spatial loops. Skewing an inner spatial loop i w.r.t. an enclosing spatial loop j with a skew factor $s_{i,j}$ involves adding $s_{i,j}$ times the loop index variable of j to the upper and lower bounds of i and subtracting the same quantity from every use of the loop index variable of i inside the loop as Wolfe described [20]. Skewing a spatial loop i w.r.t. another spatial loop j in a sibling loop nest[3] with skew factor $s_{\hat{i},j}$ is applied by adding $s_{\hat{i},j}$ to the upper and lower bounds of i and subtracting the same amount from each occurrence of the index variable of i. We call skew factors of both types of skewing for spatial loops *spatial skew factors*.

[3]Two loops or two loop nests are called sibling if they are enclosed in a common outer loop

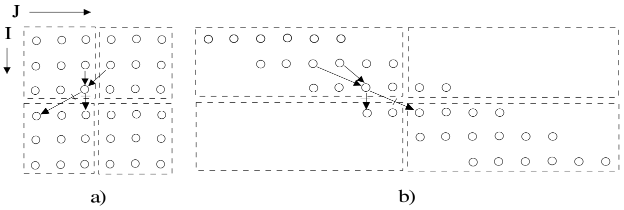

Figure 3: Skewing a spatial loop to eliminate interchange-preventing dependences.

For orthogonal recursive blocking of an iteration space[4] to be legal, the data dependences within the iteration space must be in a form so that when any dimension of the iteration space is partitioned into two halves, all true and anti- data dependences flow from the first half into the second half in lexicographical order. With this condition satisfied, the partition containing the head of the dependences can be executed in its entirety before the partition containing the dependence tails. If the condition is not satisfied, no serial ordering of the partitions will be possible.

To satisfy this condition, the distance vectors for all true and anti- dependences must contain only non-negative distances for loops whose index variables appear in array subscripts [20]. For perfect loop nests, this condition is equivalent to there being no interchange-preventing dependences. Figure 3a shows a pictorial representation of a pattern of true and anti- dependences within an iteration space. Figure 3b shows a transformed iteration space in which the interchange-preventing dependences are eliminated by skewing. For imperfect loop nests, dependences between sibling loop nests must also meet this criteria.

ComputeSpatialSkewFactors determines spatial skew factors based on dependences carried by the spatial loops and loop-independent dependences between sibling spatial loop nests enclosed in the time step loop. For each dependence $\delta^{'}$ carried by a spatial loop, $(A(g_{src}(\vec{i}))\delta^{'}A(g_{sink}(\vec{j})))$, where g_{src} and g_{sink} are subscript expressions in the source and sink references of the dependence, if $dist_i(\delta^{'}) < 0$, loop i will be skewed w.r.t. the outermost loop j satisfying $dist_j(\delta^{'}) > 0$. $dist(\delta^{'})$ is the dependence distance vector of $\delta^{'}$. For each loop independent dependence between sibling loop nests or dependence carried by a spatial loop 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, we define $dist(\delta)$ as $f_{sink}^{-1}(f_{src}(\vec{i}))$ - $\vec{i}$. If there exists k such that $dist_k(\delta) < 0$, and i and $\bar{i}$ are corresponding loops in the sibling loop nests enclosing f_{src} and f_{sink}, then loop $\bar{i}$ needs to be skewed w.r.t. loop i with a spatial skew factor $s_{\hat{i},i} = -dist_k(\delta)$.

ComputeSpatialSkewFactors(L, G, *type*)
 L: set of loops to which prismatic skewing is applied;
 G: dependence graph annotated with distance vectors.
 type: 1 for skewing w.r.t. an enclosing loop or
  2 for skewing w.r.t. a loop in a sibling loop nest

[4]We use the term orthogonal blocking to indicate that the iteration space is partitioned by splitting the index space along the (orthogonal) data dimensions.

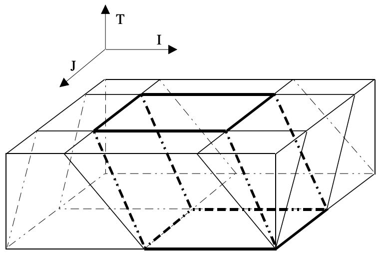

Figure 4: Prismatic skewing of a 2D domain shown in 3D space.

 $carriedDep$ = dependences carried by a spatial loop in **L**;
 $indDep$ = loop-independent dependences between
  sibling loop nest of spatial loops in **L**;
 switch (*type*)
 case 1:
  for $\delta^{'}\ (A(g_{src}(\vec{i}))\ \delta^{'} A(g_{sink}(\vec{j}))) \in carriedDep$
   $dist(\delta^{'})$ is the distance vector of $\delta^{'}$ in **G**;
   if $dist_i(\delta^{'}) < 0$
    j is the outermost loop satisfying $dist_j(\delta^{'}) > 0$;
    $s_{i,j} = max(s_{i,j}, \lceil(1-dist_i(\delta^{'}))/dist_j(\delta^{'})\rceil)$ if $s_{i,j}$ is defined, $s_{i,j} = \lceil(1-dist_i(\delta^{'}))/dist_j(\delta^{'})\rceil$ otherwise;
 case 2:
  for $\delta\ (A(f_{src}(\vec{i}))\ \delta\ A(f_{sink}(\vec{j}))) \in indDep \bigcup carriedDep$
   $dist(\delta) = f_{sink}^{-1}(f_{src}(\vec{i}))$ - $\vec{i}$;
   if there exists k such that $dist_k(\delta) < 0$, i and $\bar{i}$ are corresponding loops in sibling loop nests enclosing f_{src} and f_{sink}
    $s_{\hat{i},i} = max(s_{\hat{i},i}, -dist_k(\delta))$ if $s_{\hat{i},i}$ is defined,
    $s_{\hat{i},i} = -dist_k(\delta)$ otherwise;

3.1.2 Computing Temporal Skew Factors

Skew factors for skewing of spatial loops w.r.t. a time step loop t, called *temporal skew factors*, are determined based on data dependences carried by t so that a skewed portion of the domain iteration space at time step t can safely execute after a portion of the iteration space finishes the execution at time step $t-1$. Temporal skew factors can be represented by a vector $\vec{s}$ with each component representing a skew factor of skewing a spatial loop and corresponding spatial loops from its sibling loop nests w.r.t. the time step loop. Figure 2 shows plane and 3D views of a 2D skewing of a central prism with temporal skew factor vector $\vec{s} = (s_i, s_j)$. Suppose $[lb_i : ub_i, lb_j : ub_j]$ is the spatial cross section of the prism evaluated at time step t. At the next time step $t+1$, the spatial cross section executed is $[lb_i - s_i : ub_i - s_i, lb_j - s_j : ub_j - s_j]$. This process continues until it reaches the face of the prism defined by the upper time bound (ub_t). Boundary prisms are skewed in the same way except they are clipped by lower and upper bounds of the spatial loops. Figure 4 shows a central prism and eight different prism fragments that could be formed along boundaries with prismatic skewing, assuming no periodic boundary conditions in the spatial dimensions.

Naive skewing of spatial loops w.r.t. the time step loop can violate data dependences originally preserved by the sequen-

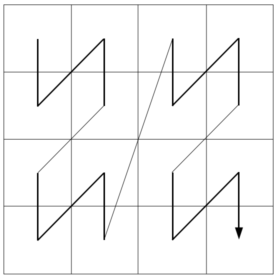

Figure 5: 2D Morton curve.

tial order of time steps. Prismatic skewing preserves dependences within a prism by incrementally traversing time steps and walking through the original portion of each prism and its skewed portions in the original lexicographical ordering or some other legal ordering. With properly chosen skew factor vector, prismatic skewing also preserves dependences between prisms and carried by the time step loop.

For loops with periodic boundary conditions, true dependences exist between iterations at their upper bounds and those at the lower bounds. Execution of those boundary iterations must be delayed until the values they need are updated. A *reverse skewing* in the boundary prisms is needed. Reverse skewing with a temporal skew factor $\vec{rs}$ clips the boundary prisms along the dimension of loop i with $L_i + rs_i(t - lb_t)$ and $U_i + rs_i(t - lb_t)$, where L_i and U_i are lower and upper bounds of loop i, rs_i is the component of $\vec{rs}$ associated with loop i, and t and lb_t are the current time step and lower bound of the time step loop.

ComputeTemporalSkewFactors calculates temporal skew factors for each of the spatial loops in **L** based on dependences carried by the time step loop t. All temporal skew factors are initially set to zero. For each dependence δ' carried by t, $A(f_{src}(\vec{i}))\ \delta'\ A(f_{sink}(\vec{j}))$, where f_{src} and f_{sink} are subscript expressions of the source and sink references of the dependence, the temporal skew factor for spatial loop l is updated by $s_l = max(s_l, -dist_l(\delta'))$. $dist(\delta')$ is defined as $f_{sink}^{-1}(f_{src}(\vec{i}))$ - $\vec{i}$. If l has periodic boundary conditions, the temporal skew factor for reverse skewing is calculated $rs_l = max(rs_l, min(dist_l(\delta'), U_l - L_l + 1 - dist_l(\delta')))$, where U_l and L_l are upper and lower bounds of loop l. For periodic boundary conditions, we used code sinking [21] to move boundary computations into corresponding spatial loops, if they are originally peeled off the main loop body, by adding proper conditions.

ComputeTemporalSkewFactors(L)
  L: set of loops to which prismatic skewing is applied;
  for each spatial loop $l \in \mathbf{L}$
    initialize skew factors $s_l = rs_l = 0$;
  for each dependence δ' carried by t
    $s_l = max(s_l, -dist_l(\delta'))$;
    if l has periodic boundary conditions
      $rs_l = max(rs_l, min(dist_l(\delta'), U_l - L_l + 1 - dist_l(\delta')))$.

3.2 Recursive Partitioning

For cached data to be reused in multiple time steps, each prism needs to be small enough to fit in the relevant cache. In this section, we present a recursive partitioning framework for the prismatic skewing and address issues of choosing base block size and recursive ordering.

3.2.1 Determining Base Block Size

If locality were the only concern, it would be sufficient to partition the iteration space until each block consists of a singleton. The overhead of approximately $n_1 \times n_2 \times ... \times n_m$ calls to leaf routines will negate the locality advantages. Instead, we choose to cut off the recursion at a base case size that is guaranteed to fit in the primary cache of any system of interest. **ComputeBaseSize** illustrates the computation of the base block size, $BLOCK$, based on the data volume of an unclipped prism and target primary cache size. The algorithm first computes the data space accessed for each array A in the tiled time step loop by taking the union of the data spaces touched within a single time step t, $D_{A,t}(BLOCK)$, denoted $vol_{A,t}(BLOCK)$. If $\vec{s}$ is the skew factor vector of the prismatic skewing, each time step will touch $\sum_{i=1}^{dim(\vec{s})}(\frac{s_i}{BLOCK} * vol_{A,t}(BLOCK))$ locations in the array that are not already touched by previous time steps. To avoid making $BLOCK$ too small, we execute $\frac{BLOCK}{s_{max}}$ time steps in the prism, where $s_{max} = max\{s_i \mid 1 \leq i \leq dim(\vec{s})\}$. The data volume touched in array A across $\frac{BLOCK}{s_{max}}$ steps is $vol_{A,t}(BLOCK) * (1 + \sum_{i=1}^{dim(\vec{s})} \frac{s_i}{s_{max}})$. Taking the sum of data volumes for all array variables touched in the prism yields the total data volume. We choose $BLOCK$ so that the total data volume does not exceed the effective size of the primary cache cs. For cases in which the total number time steps is smaller than $\frac{BLOCK}{s_{max}}$, we recompute the $BLOCK$ with the smaller number.

ComputeBaseSize(L, $\vec{s}, cs$**)**
  L: set of loops to which recursive skewing is applied;
  $\vec{s}$: skew factor vector;
  cs: effective cache size;
  $s_{max} = max\{s_i \mid 1 \leq i \leq dim(\vec{s})\}$;
  for each array referenced inside loop t, $A \in array(t)$
    determine data set of A accessed by each t step:
      $D_{A,t}(BLOCK) = \bigcup_{r \in ref_A(t)} D_{A,t}^{r}(BLOCK)$;
    compute data volume $vol_{A,t}(BLOCK)$;
    compute data volume $vol_A(BLOCK)$ in $\frac{BLOCK}{s_{max}}$
      steps of t: $vol_A(BLOCK) =$
      $vol_{A,t}(BLOCK) * (1 + \sum_{i=1}^{dim(\vec{s})} \frac{s_i}{s_{max}})$;
    $vol_{total}(BLOCK) = \sum_{A \in array(t)} vol_A(BLOCK)$;
  select base block size $BLOCK$ so that
    $vol_{total}(BLOCK) \leq cs$;
  if $(\frac{BLOCK}{s_{max}} > ub_t - lb_t + 1)$
    recompute base black size $BLOCK$ so that
    $\sum_{A \in array(t)}(vol_{A,t}(BLOCK) *$
      $(1 + \frac{\sum_{i=1}^{dim(\vec{s})} s_i * (ub_t - lb_t + 1)}{BLOCK})) \leq cs$;

3.2.2 Partitioning the Computation

Dependences between prisms that are loop-independent, carried by spatial loops, or carried by the time step loop can all prevent naive partitioning. The skewing transformations described in Section 3.1 are used to transform these dependencies so they are all in a forward direction when the prisms are executed in either a lexicographic or an appropriate recursively-defined execution order. The latter have the advantage over lexicographic orders in that they foster temporal reuse of data from sibling prisms that are resident in secondary and higher levels of the cache.

To define an recursive execution order with good inter-prism reuse, we use bisection and Morton ordering. Figure 5 shows a 2-D Morton curve. **ChoosePartitioningandOrder** shows the process of recursive partitioning and execution order of the result subspaces. First, the m dimensional spatial iteration space $n_1 \times n_2 \times ... \times n_m$ is bisected recursively along the longest dimensions until dimensions of each subspace $n'_1 \times n'_2 \times ... \times n'_m$ are balanced, *i.e.* $\frac{n'_{max}}{n'_{min}} < 2$, where n'_{max} and n'_{min} are the longest and shortest dimensions of the subspace. Each subspace is further 2^m-way partitioned into subspaces of size $\frac{n'_1}{2} \times \frac{n'_2}{2} \times ... \times \frac{n'_m}{2}$ until the length along any one axis does not exceed the base block size $BLOCK$. Prismatic skewing is then applied at the base level as discussed in the previous section. Cut points are carefully chosen to be at primary cache line boundaries to improve reuse. cls is the primary cache line size.

ChoosePartitioningandOrder(L)
    L: set of loops to which recursive skewing is applied;
    $minL = \{i \mid ub_i - lb_i = min_{k \in \mathbf{L}}(ub_k - lb_k)\}$;
    $maxL = \{i \mid ub_i - lb_i = max_{k \in \mathbf{L}}(ub_k - lb_k)\}$;
    if $(ub_{maxL} - lb_{maxL} < 2 * (ub_{minL} - lb_{minL}))$
        use $2^{nestLevel(\mathbf{L})}$-way partitioning and Morton
            ordering for the loops in **L**;
        set ranges of i ($i \in \mathbf{L}$) after partitioning:
            $[lb_i : \lceil \frac{\lceil (lb_i + ub_i)/2 \rceil}{cls} \rceil * cls]$,
            and $[\lceil \frac{\lceil (lb_i + ub_i)/2 \rceil}{cls} \rceil * cls + 1 : ub_i]$;
    else
        use 2-way bisection and lexicographical ordering;
        set ranges of loop $maxL$ after partitioning:
            $[lb_{maxL} : \lceil \frac{\lceil (lb_{maxL} + ub_{maxL})/2 \rceil}{cls} \rceil * cls]$
            and $[\lceil \frac{\lceil (lb_{maxL} + ub_{maxL})/2 \rceil}{cls} \rceil * cls + 1 : ub_{maxL}]$;

3.3 Inter-procedural Analysis

Typically, the temporal and spatial loops in large applications are spread across multiple routines. Inter-procedural analysis is essential if RPTS and related techniques are to be applied to these codes. In our current interprocedural framework, we assume indices of the spatial loops are only used locally. For cases that spatial loop indices are used globally or passed at call sites, selective inlining substitution or loop embedding [12] is used. Prismatic time skewing can be applied w.r.t. multiple time step loops as long as they are not imperfectly nested. For time step loops that are imperfectly nested, only the innermost is transformed. Figure 6 shows a code segment with a time step loop and five-point stencil computation in different routines.

Our inter-procedural analysis framework operates in two phases. In the first phase, we identify time step loops by a bottom-up pass followed by a top-down pass through the call graph. In the bottom-up pass, **IdentifyNonIterativeLoop** marks a scalar variable as *non-iterative* if it is used to compute a subscripted reference. At each call site, if a dummy parameter is non-iterative, we mark the scalar

```
do t = 1, U_t
   call foo(a, b)
   do j = 1, n
      do i = 1, m
         b(i, j) = a(i, j)
...
subroutine foo(a, b)
...
do j = 2, n − 1
   do i = 2, m − 1
      a(i, j) = (b(i, j) + b(i − 1, j) + b(i + 1, j)
               + b(i, j − 1) + b(i, j + 1))/5
```

Figure 6: An example with temporal and spatial loops in separated routines.

variables used in the corresponding actual parameter as non-iterative. **MarkNonIterativeVars** marks the variable as *non-iterative* if it is a loop index variable, an induction variable, a formal parameter of a routine, or a global variable. It also recursively checks the variables in right-hand-side of the reaching definitions of *use*. In the top-down pass, **SelectLoopsForSkewing** marks a loop variable t as *iterative* if it is not non-iterative and select a non-iterative loop i for skewing if there is no data dependences with non-constant distance at the level of loop i or dependence δ such that $A(f_{src}(\vec{i})\delta A(f_{sink}(\vec{j})$ and $i \in f^{-1}_{sink}(f_{src}(\vec{i}))$ - $\vec{i}$ or $j \in f^{-1}_{src}(f_{sink}(\vec{j}))$ - $\vec{j}$. At each call site, we propagate this information from caller to callee.

IdentifyNonIterativeLoop(C)
    C: a call graph of the original code;
    for each routine **P** of **C** in bottom-up order
        for each subscripted reference *ref* in **P**
            for each use *use* of a scalar variable *var* in *ref*
                MarkNonIterativeVars(*use*);
        for each call site c
            if callee is marked as *iterativeExist*
                mark enclosing loops of c as non-iterative;
                mark **P** as *iterativeExist*;
            if a dummy parameter *dpar* is *non-iterative*
                for each use *use* of scalar variables in the
                    corresponding actual parameter *apar*
                    MarkNonIterativeVars(*use*);
        if there is a loop that is not marked as non-iterative
            remark its enclosing loops as non-iterative if they
                haven't been marked;
            mark **P** as *iterativeExist*.

SelectLoopsForSkewing(C, G, *cfg***)**
    C: a call graph of the original code;
    G: a dependence graph annotated with distance vectors.
    cfg: a control flow graph of each routine;
    initialize loop set $\mathbf{L} = \emptyset$;
    attach **L** to the main routine;
    for each routine **P** of **C** in top-down order
      for each **L** attached to **P**;
        for each node of *cfg* of **P** in top-down forward order
            if the node is a loop t and it is not non-iterative
                mark t as iterative;
                $\mathbf{L} = \mathbf{L} \cup \{t\}$;

$\mathbf{L} = \mathbf{L} \cup \{i \mid i$ is a loop enclosed in $t\}$;
$\mathbf{L} = \mathbf{L} - \{i \mid i$ is a loop enclosed in t and there is a dependence δ such that $dist_{level(i)}(\delta)$ is not constant$\} - \{i, j \mid i$ and j are loops enclosed in t and there is a dependence δ such that $A(f_{src}(\vec{i}))\ \delta\ A(f_{sink}(\vec{j}))$ and $i \in f_{sink}^{-1}(f_{src}(\vec{i})) - \vec{i} \bigvee j \in f_{src}^{-1}(f_{sink}(\vec{j})) - \vec{j})\}$;
if the node is call site c
propagate $\mathbf{L}$ to the callee as $\mathbf{L}_c$.

During the second phase, a bottom-up traversal of the call graph computes spatial skew factors and temporal skew factors for each of the spatial loops, estimates data volume for a prism, and summarizes and propagates the information upward along the graph. Preskewing spatial loops w.r.t. enclosing spatial loops is performed if it is necessary. Dependence information is updated after preskewing. Spatial skew factors are computed with **ComputeSpatialSkewFactors** and finalized as $\hat{s_i}$ for each spatial loop i by $\hat{s_i} = max_{j \in \mathbf{L}} \hat{s_{i,j}}$ and $\hat{s_i} = max(\hat{s_i}, \hat{s_{i,j}} + \hat{s_j})$ if there exists spatial loop j such that $\hat{s_{i,j}} \neq 0$. Spatial loops are skewed w.r.t. each other with the finalized spatial skew factor. After that, temporal skew factors are computed with **ComputeTemporalSkewFactors** and adjusted by choosing the maximum value of the skew factors of corresponding spatial loops in sibling loop nests. Computed skew factor and base block size for recursive skewing are finally propagated downward along the call graph. Information on skew factors and blocking size may different for different call sites of a callee.

For the example shown in Figure 6, **IdentifyNonIterativeLoop** first marks loops i and j in subroutine *foo* and the caller as non-iterative. Then **SelectLoopsForSkewing** identifies loop t as iterative and adds loop t and spatial loops i and j after the call site into loop set $\mathbf{L}$ attached to the caller. Loop set attached to *foo* is computed as t, i, j, where i and j are the spatial loop inside *foo*. During the second phase, spatial skew factors for skewing the spatial loop i and j in the caller w.r.t. the other spatial loops in $\mathbf{L}$ are 1 according to **ComputeSpatialSkewFactors**. After skewing the spatial loops with spatial skew factor, temporal skew factors are computed to be (2,2) with **ComputeTemporalSkewFactor**.

3.4 Code Generation

Applying prismatic time skewing to a program may involve preskewing spatial loops w.r.t. enclosing spatial loops, skewing spatial loops w.r.t. other spatial loops in their sibling loop nests, and skewing spatial loops w.r.t. time step loops. We have briefly described code generated by applying preskewing and skewing spatial loops w.r.t. other spatial loops in sibling loop nests in Section 3.1.1. In this subsection, we discuss how to generate code for a prism and apply recursive blocking on top of it.

Once we have computed the temporal skew factor vector $\vec{s}$ and the prism size, code generation for a prism consists of following three steps. First, clip the range of the original t loop to $[lb_t : ub_t]$. Then, for each spatial loop i that needs to be skewed w.r.t. the time step loop, the original lower and upper bounds $[L_i : U_i]$ are replaced with $[max(L_i, lb_i - s_i * (t - lb_t)) : min(U_i, lb_i - s_i * (t - lb_t) + BLOCK_i - 1)]$, where $BLOCK_i$ is the base block size for loop i, and lb_i is the lower bound of loop i at the time step $t = lb_t$. Finally, insert if-conditions for statements that are not originally enclosed by any n of the skewed spatial loops to preserve loop independent dependence between the statements and the loops before and after them. n is the number of blocked dimensions in spatial domain. For loops with periodic boundary conditions, the prisms at the lower boundaries will be clipped by $L_i + rs_i * (t - lb_t)$ instead of L_i. The prisms being clipped at the lower boundaries will be wrapped with the matching prisms at the upper boundaries.

CodeGenForPrism($\mathbf{L}$, $\vec{s}, \vec{rs}, \vec{b}, lb_t, ub_t$)
$\mathbf{L}$: nested loops on which prismatic skewing is applied;
$\vec{s}$: skew factor vector;
$\vec{rs}$: skew factor vector for reverse skewing;
$\vec{b} = (BLOCK_1, BLOCK_2, ..., BLOCK_l)$: the ranges along each dimension of l spatial loops;
lb_t, ub_t: lower and upper bounds of the time steps we are exploiting temporal reuses.
modify the range of the t loop to $[lb_t : ub_t]$;
for each spatial loop i in $\mathbf{L}$ with bounds L_i, and U_i
if i has no periodic boundary conditions
modify the loop range to
$[max(lb_i - s_i * (t - lb_t), L_i)$
$: min(lb_i + BLOCK_i - 1 - s_i * (t - lb_t), U_i)]$;
else
create a loop i_{bound} before loop i with loop range
$[max(lb_i - s_i * (t - lb_t), U_i + 1)$
$: min(lb_i - s_i * (t - lb_t) + BLOCK_i - 1,$
$U_i + rs_i * (t - lb_t))]$;
insert statement $i = mod(i_{bound}, U_i) + L_i - 1$ as the first statement of the loop;
copy the original loop body next to the statement;
modify the loop range of i to
$[max(lb_i - s_i * (t - lb_t), L_i + rs_i * (t - lb_t))$
$: min(lb_i + BLOCK_i - 1 - s_i * (t - lb_t), U_i)]$;
for each statement that is not originally enclosed by any n of the spatial loops that are skewed w.r.t. t
insert proper if-conditions to preserve dependences.

As an example, we present the following 2D five point stencil code. Without loss of generality, we normalized loop bounds in the code. Array a is a double precision array of size 1K× 1K. U_t is the upper bound of the time step loop. U_i, and U_j are the upper bounds of the two spatial loops.

```
do t = 1, U_t
  do i = 1, U_i
    do j = 1, U_j
      a(j,i) = 0.125 * (a(j+1,i) + a(j-1,i)
               + a(j,i+1) + a(j,i-1) + 4 * a(j,i))
```

There are two classes of dependences in the code: dependences carried by the time step loop and dependences carried by spatial loop i or j. Their distance vectors are (1, -1, 0), (0, 1, 0), (1, 0, -1), (0, 0, 1), and (1, 0, 0). Each point is updated based on its value at the previous time step and the values of its four neighbors. Of the neighbors, two are computed in the previous time step, while the other two are updated in the current time step. **ComputeSkewFactors** skews the spatial loops i and j w.r.t. the time step loop with the vector (1,1). Since the boundary condition is non-periodic, the reverse skew factor vector $\vec{rs}$ is $\vec{0}$. Since there is only one spatial loop nest, skewing or alignment between sibling spatial loops is not needed. The code for the execution of a prism is shown below.

```
do t = lb_t, ub_t
   do i = max(lb_i - t + lb_t, 1),
          min(lb_i + BLOCK_i - 1 - t + lb_t, U_i)
      do j = max(lb_j - t + lb_t, 1),
             min(lb_j + BLOCK_j - 1 - t + lb_t, U_j)
         a(j,i) = 0.125 * (a(j+1,i) + a(j-1,i)
                  + a(j,i+1) + a(j,i-1) + 4 * a(j,i))
```

Any legal ordering can be used to iterate through the prisms. Lexicographical ordering can be implemented by creating an outer loop i_{out} for each spatial loop i to traverse the space of prisms, replacing the lower bound of i, lb_i, at the time step $t = lb_t$ with i_{out}, and creating an outer time step loop t_{out} to iterate the time blocks. In general, the outer loop will have to iterate through additional (clipped) prisms besides $\lceil \frac{U_i - L_i + 1}{BLOCK_i} \rceil$ to cover the entire range of i. **ComputeExtraPrisms** calculates the number of prisms needed additionally based on base block size and skew factors.

ComputeExtraPrisms(L, $\vec{s}, \vec{rs}, \vec{b}$)
- **L**: the nested loops the prismatic skewing is applied;
- $\vec{s}$: skew factor vector;
- $\vec{rs}$: skew factor vector of reverse skewing for loops with periodic boundary conditions;
- $\vec{b} = (BLOCK_1, BLOCK_2, ..., BLOCK_l)$: the ranges along each dimension of l spatial loops;
- **for** each spatial loop i in **L**
  - **if** i has no periodic boundary conditions
    $r_i = \lceil \frac{(ub_t - lb_t) * s_i - (\lceil \frac{U_i - L_i + 1}{BLOCK_i} \rceil * BLOCK_i - (U_i - L_i + 1))}{BLOCK_i} \rceil$;
  - **else**
    $r_i = \lceil \frac{(ub_t - lb_t) * (s_i + rs_i) - (\lceil \frac{U_i - L_i + 1}{BLOCK_i} \rceil * BLOCK_i - (U_i - L_i + 1))}{BLOCK_i} \rceil$;
- return $(r_1, r_2, ..., r_l)$;

Code generation to add recursive blocking to prismatic skewing and to traverse the prisms in, for instance, Morton order has four steps. The original time step loop and its loop body is replaced with a recursive call parameterized by the lower and upper bounds of each spatial loop to be skewed. As in code generation for non-recursive prismatic skewing, the range of each spatial loop may need to be extended or clipped. The code generator then creates a recursive routine including code for a prism at base level as discussed in the previous subsection. For the base case, the lower bound of loop i in the code for a prism, lb_i, is generated using the passed lower bound. At internal nodes of the recursion, the synthesized routine calls itself recursively with reduced bounds and using Morton ordering to satisfy the dependences left after skewing. Finally, a generated outer loop t_{out} iterates through the layers of prisms along the time dimension.

CodeGenForRecursiveSkewing(L, $\vec{s}, \vec{rs}, \vec{b}, lb_t, ub_t$)
- **L**: nested loops the prismatic skewing is applied;
- $\vec{s}$: skew vector;
- $\vec{rs}$: reverse skew vector for loops with periodic boundary conditions;
- $\vec{b} = (BLOCK_1, BLOCK_2, ..., BLOCK_l)$: the ranges along each dimension of l spatial loops;
- lb_t, ub_t: lower and upper bounds of the time steps we are exploiting temporal reuses.
- compute extra prisms to iterate:
  $(r_1, ..., r_l) =$ **ComputeExtraPrisms(L**, $\vec{s}, \vec{rs}, \vec{b}$);
- replace the original loop **L** with a recursive call and pass parameters L_i and $U_i + r_i * BLOCK_i$ as lower and upper bounds of spatial loop i which needs to be skewed w.r.t. the time step loop;
- create recursive routine with code generated by
  CodeGenForPrism(L, $\vec{s}, \vec{rs}, \vec{b}, lb_t, ub_t$)
  for a prism at base level
- insert recursive calls at upper level as described in
  ChoosePartitioningandOrder;
- create an outer loop t_{out} iterating multiple time blocks;

Applying time skewing to convergence loops of iterative methods raises several problems for which we currently have no automatable solution. Convergence is usually tested on every iteration. If the method is stable then it may be acceptable to test convergence only at the end of the prism computation, *i.e.* every $ub_t - lb_t + 1$ time steps. If it is important to exactly replicate the behavior of a non-skewed execution, speculative execution [19] could be used to roll back to the first converged iteration. This is done by checkpointing the computation at between layers of prisms in the "convergence" dimensions. If there is early convergence, the execution could be replayed from the checkpoint using the original execution order with convergence tests on every time step. Many iterative methods compute one or more global inner products on every iteration and use the computed value(s) in the next iteration.

4. EXPERIMENTAL RESULTS

We are currently implementing recursive prismatic time skewing in the Rice dHPF compiler infrastructure[1, 2]. To evaluate the effectiveness of the ideas, we manually applied the analyses and transformations to a set of benchmark programs. Due to space limitation, we only include results of two numerical kernels, Jacobi and SOR, one hydrodynamic kernel from the Livermore benchmarks; and SMG988 [6], a semi-coarsening multi-grid benchmark written in C. For repeatability, experiments for the three kernels were performed on an SGI O2 workstation with a 195 MHz MIPS R10K processor, 256MB main memory, a 32KB 2-way on-chip L1 data cache, a 1MB 2-way unified L2 cache and a 64-way 512KB TLB. SMG98 requires more memory than is available on the workstation, so experiments were performed on one processor of a 16-processor SGI Origin 2000 with 300 MHz MIPS R12K processors, 10GB main memory, and 8MB L2 caches. The kernels were compiled `-O3` with the MIPSpro F77 compiler V7.3.1.1 and SMG98 was compiled `-O2` with the MIPSpro C compiler. Hardware performance counters and *perfex* were used to measure execution time and cache misses. We measured L1 and L2 cache misses in separate runs to avoid multiplexing the performance counters. Each experiment was repeated 5 times. Variations were small and the average measurements are presented. Execution time for SMG98 is not reported because we were not able to run the code in dedicated mode on the SGI Origin.

We compared our transformed codes with hand implemen-

Table 5: L1(L2) Cache misses (in millions) of `LIVERMORE 18`.

Test Cases	*400*	*600*	*800*	*1000*	*1200*
`orig`	53.2(21.3)	116.(50.8)	547.(84.6)	317.(143.)	458.(217.)
`SSB`	49.9(11.6)	118.(25.6)	560.(44.0)	262.(61.9)	469.(152.)
`Song & Li`	59.6(1.51)	152.(4.47)	563.(10.5)	438.(30.1)	625.(82.7)
`RPTS`	16.0(1.02)	69.6(2.95)	202.(3.82)	140.(6.19)	163.(17.0)

Table 1: L1(L2) Cache misses (in millions) of `SOR`.

Test Cases	*512*	*1024*	*2048*	*4096*
`orig`	4.82(2.29)	29.3(8.78)	180.(35.1)	312.(141.)
`SSB`	4.99(2.18)	28.5(8.60)	158.(34.1)	324.(139.)
`Song & Li`	4.83(0.08)	29.1(0.35)	76.7(2.56)	455.(18.0)
`RPTS`	0.49(0.07)	2.07(0.28)	12.5(1.18)	48.3(5.63)

Table 2: Sequential execution time (in million cycles) of `SOR`.

Test Cases	*512*	*1024*	*2048*	*4096*
`orig`	403	1761	6625	27569
`SSB`	412	1586	6370	27778
`Song & Li`	106	446	1961	9160
`RPTS`	94	398	1603	6929

tations of the techniques described in recent papers by Song and Li [18, 19] and Wonnacott [22]. These versions of the benchmarks are denoted "new tiling" and "time skewing", respectively. For new tiling we used odd-even duplication, loop fusion, and forward substitution since this combination was reported to produce the best results. For SOR we used in-place computation in all implementations to save storage and extra copying. The choice of *nstep* (tile size) for each problem instance was chosen by picking the best performer in a range from 10% to the full size of the secondary cache. Although 0.3 of the physical cache size was cited as the optimal size, we found that other sizes often worked better.

We produced a test code using Wonnacott's time skewing only for Jacobi, since the other kernels have loop carried dependences in all of the spatial dimensions. We used *blocked storage mapping* because it was clear that it would perform best for a Jacobi four point stencil and it was not clear how to implement *time skewed storage mapping*. The tile sizes s used for time skewing are the ones that performed best for each problem size.

To better evaluate how much performance improvement is due solely to explicit spatial skewing and blocking, and to better evaluate the additional benefit of time skewing and recursive blocking, we also measured execution time and cache misses for versions of the numerical kernels and the Livermore benchmark that use explicit spatial skewing and blocking. These are denoted "SSB" in the tables shown below. Blocking sizes used for the data presented are the ones that performed best.

4.1 SOR

SOR (successive over-relaxation) uses a two-dimensional five point Gauss-Seidel relaxation step. The number of time steps was fixed at 64 and we used matrix dimensions of N = 512, 1024, 2048, and 4096. For the new tiling code the *nstep* for these four data sizes were 60, 30, 23, and 15 because they produced better cache and overall performance than the other choices [19]. To reduce cache conflict misses, the MIPSpro compiler pads arrays in both the original program and in the new tiling version, but it failed to do so for the prismatic skewing version. We therefore applied the same padding increment manually. In the prismatic skewing case we turned off prefetching because it was ineffective without manifest knowledge of the loop bounds at the base case of the recursion. Automatic blocking was turned off because recursion achieves the same effect. These two optimizations were effective for the other two versions, so they were left on for those experiments. The base block size for prismatic skewing is 21 across all four data sizes. Results presented in Table 1 and Table 2 indicate that blocking spatial loops alone cannot effectively reduce cache misses and execution time. Both Song and Li's version and ours significantly improved cache and overall performance by exploiting data reuses across time steps. By using multidimensional time skewing and recursive blocking, primary cache misses are reduced by factors of 6 to 14 over new tiling and secondary cache misses are decreased by about 14% to a factor of 3 better than new tiling. Execution time is improved by about a factor of 4 over the original program and by 12% to 32% over new tiling.

Table 3: L1(L2) Cache misses (in millions) of `JACOBI`.

Test Cases	*512*	*1024*	*2048*	*3072*
`orig`	19.0(8.79)	98.0(34.2)	377.(137.)	867.(309.)
`SSB`	12.2(4.15)	90.3(16.7)	433.(67.3)	803.(152.)
`Song & Li`	9.80(0.21)	59.8(0.89)	247.(6.36)	523.(35.0)
`Wonnacott`	1.03(0.33)	6.69(1.76)	56.9(8.27)	60.7(21.2)
`RPTS:minmem`	1.37(0.04)	5.57(0.46)	27.8(1.61)	50.7(4.13)
`RPTS:nocopy`	1.39(0.13)	6.39(0.76)	25.5(2.43)	59.6(5.21)

Table 4: Sequential execution time (in million cycles) of `JACOBI`.

Test Cases	*512*	*1024*	*2048*	*3072*
`orig`	1462	5806	23501	52933
`SSB`	821	3343	13542	30637
`Song & Li`	138	575	2729	8905
`Wonnacott`	188	987	4342	9781
`RPTS:minmem`	148	661	2752	6006
`RPTS:nocopy`	95	409	1601	3816

4.2 2D Jacobi

2D Jacobi uses Jacobi relaxation with a four point stencil. There are no dependences carried on the spatial loops. We

Table 6: Sequential execution time (in million cycles) of `LIVERMORE 18`.

Test Cases	*400*	*600*	*800*	*1000*	*1200*
`orig`	3694	8742	18952	24582	37038
`SSB`	2275	5014	12148	13255	29659
`Song & Li`	1000	2438	8496	9688	20453
`RPTS`	841	2095	4751	6136	10785

Table 7: Cache misses (in millions) of `SMG98`.

Code	*L1 misses*		*L2 misses*	
	orig	*opt*	*orig*	*opt*
`residual`	4327	3114	457	232
`cyclic_reduction`	4187	3220	506	132
`total`	9406	7388	1100	502

used 64 iterations and problem arrays that were 512, 1024, 2048, and 3072 elements on a side. Additional space is necessary to preserve old values located within each tile and on the boundaries between tiles. We only used the minimum storage to keep the old values until they are not needed anymore. The amount of additional storage used depends on dependence distances and the number of time steps we are exploiting data reuses. For the experiments we performed, we are able to reduce storage to 75M and 2.5M bytes for 512 by 512 and 1K by 1K problems from 144M and 4M compared with other three versions. However cost of storage saving is extra copyings to preserve old values. To understand this cost of extra copying, we also include performance data for an implementation without copying, but using as much storage as the other three non-RPTS versions. The two RPTS versions are marked as RPTS:nocopy and RPTS:minmem in Table 3 and 4.

For new tiling *nstep* was set to 50, 12, 14, and 6 for data sizes 512, 1024, 2048, and 3072, respectively. The tile sizes used for time skewing are 28, 12, 12, and 12, respectively. We also applied optimizations to eliminate most of the *mod* and *div* in the transformed code shown in [22]. The base block size for prismatic skewing was 16. As in the case of SOR, we used manual array padding and turned off automatic blocking and prefetching options in the compiler.

As Table 3 shows, spatial skewing and blocking (SSB) is able to reduce secondary cache misses by about a factor of 2, and execution time by 73% to 78% over the original version. The improvement comes from data reuses across spatial loop nests. Other versions further improve cache and overall performance to different extent by exploiting reuses across time steps in different ways. The time skewing version exhibits very high L1 cache reuse, while the new tiling effectively reduces L2 cache misses in most cases. RPTS:minmem achieves better cache reuses than these versions in all cases except one. For the 512 by 512 problem, our L1 cache misses figure is about 35% larger than that of time skewing. We believe that this is because our transformations limit the ability of the MIPSpro compiler to perform certain optimizations and that this becomes significant for this small problem. Execution time of RPTS:minmem is as much as 15% longer than that of new tiling for small problem size because of the cost of extra copyings. This advantage disappears for larger problem sizes; execution time is about 48% shorter than that of new tiling for 3K by 3K problem. RPTS:nocopy, the version without copying, achieves best overall performance in all cases with a moderate increase of cache misses compared with RPTS:minmem. We are currently investigating the gap between two RPTS versions and should be able to shorten it by improving management of copyings. Overall, RPTS shows an improvement in execution time compared with the native compiler by factors between 13.87 to 15.33 and compared to time skewing and new tiling by up to 133%. See Table 4.

4.3 Livermore 18

Livermore 18 is a hydrodynamic kernel code. The programs were run for 64 iterations on arrays of 400, 600, 800, 1000, and 1200 elements on a side. Blocking with spatial skewing is able to substantially reduce secondary cache misses, resulting in a 25% to 85% overall performance improvement over the original code. However, there is still a significant amount of data reuse across time steps. New tiling significantly outperforms both of the non-time-skewed versions. The tiling sizes *nstep* were 17, 10, 8, 4, and 2, respectively, for each of the problem sizes. RPTS used 16 as the size of the base block. Array padding was not used since the MIPSpro compiler did not add padding and cache conflict misses do not significantly affect performance. Compared with both the original and new tiling versions of the program, RPTS reduced L1 cache misses by 67% to 382%. Compared to the new tiling version, L2 misses are reduced by 48% to 487%. See Table 5. Execution time improves by factors between 3.43 and 4.39 compared to the original code and by 16% to 90% compared to the new tiling version. See Table 6.

4.4 SMG98

SMG98 [6] is a semi-coarsening multi-grid benchmark program that is approximately 21000 lines of C source code. It is a memory intensive code. For 3D problems, each data point uses roughly 54 doubles. We ran for 5 iterations with a data size of 128 × 128 × 128 for the working arrays and vectors. The program performs two sweeps of red/black plane(line) pre-relaxation and post-relaxation in each multi-grid V-cycle. Because data dependences exist at each level of the multi-grid, RPTS was applied only to the y and z dimensions in the pre-relaxation sweeps. Base block size is chosen to be 4 × 4. We transformed two routines in the application, the 1D direct solver `cyclic_reduction` and the residual computation routine `residual`. For the chosen data size, each of these routines accounts for 30% to 40% of primary cache misses, secondary cache misses, and execution time in the original program. Inter-procedural analysis was essential for identify the iterative loop and the spatial loops for prismatic skewing, and for propagating the skewing factor. The transformed code reuses data across both of the red-black relaxation sweeps, and within and between the down and up phases of the V-cycle. This reduces the total L1 cache misses by 27% and L2 cache misses by 119% as shown in Table 7.

5. FINAL COMMENTS

We describe an integrated approach for improving multi-level memory hierarchy performance in large-scale scientific applications. Recursive prismatic time skewing (RPTS) improves temporal locality at three levels: within a block for a single time step, across multiple time steps within a prism, and between prisms. An inter-procedural analysis and code generation framework enables us to apply these techniques to some kinds of large-scale scientific application. Experimental results indicates that the technique can significantly outperform previously published methods techniques on kernel benchmarks, and it improves memory hierarchy performance of part of a real application.

While RPTS can be applied to one class of application, increasing temporal reuse is an important issue for a much wider range of program. In particular, iterative methods with convergence tests and/or global inner products computed on every iteration cannot be automatically transformed by our techniques. Either semi-automatic methods or the development of new algorithms are indicated in these cases.

Acknowledgments

We would like to thank Charles Koelbel and the anonymous referees for their valuable comments and suggestions. This research was supported in part by NCSA under National Science Foundation cooperative agreement ACI-9619019, the DOE ASCI Program under research subcontract B347884, 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.

6. REFERENCES

[1] V. Adve, G. Jin, J. Mellor-Crummey, and Q. Yi. High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes. In *Proceedings of SC98: High Performance Computing and Networking*, Orlando, FL, Nov 1998.

[2] V. Adve and J. Mellor-Crummey. Using Integer Sets for Data-Parallel Program Analysis and Optimization. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

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

[4] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *Proceedings of SC'00: High Performance Networking and Computing*, Dallas, TX, Nov. 2000.

[5] N. Ahmed and K. Pingali. Automatic generation of block-recursive codes. In *Proceedings of the Euro-Par2000*, Munich, Germany, Aug. 2000.

[6] P. N. Brown, R. D. Falgout, and J. E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM J. Sci. Comput*, 21(5):1823–1834, 1999.

[7] 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.

[8] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing '92*, Minneapolis, MN, Nov. 1992.

[9] 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.

[10] 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.

[11] J. Frens and D. Wise. Auto-blocking matrix multiplication or tracking blas3 performance from source code. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 206–216, Las Vegas, NV, June 1997.

[12] M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*, Albuquerque, NM, Nov. 1991.

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

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

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

[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] H. Prokop. Cache-oblivious algorithms. Master's thesis, Department of Electrical Engineering, MIT, June 1999.

[18] Y. Song and Z. Li. A compiler framework for tiling imperfectly-nested loops. In *Proceedings of the Twelfth International Workshop on Languages and Compilers for Parallel Computing*, La Jolla, CA, Aug. 1999.

[19] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[20] M. J. Wolfe. Loop skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, Aug. 1986.

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

[22] D. Wonnacott. Time skewing: A value-based approach to optimizing for memory locality. Submitted for publication.

[23] Q. Yi, V. Adve, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, Vancouver, Canada, June 2000.