Rutgers University Department of Computer Science Technical Report 388
Submitted 24 Jun 1999

# Time Skewing for Parallel Computers *

David Wonnacott
Haverford College
Haverford, PA 19041
davew@cs.haverford.edu

Revised to May 17, 1999

**Abstract**

Time skewing is a compile-time optimization that can provide arbitrarily high cache hit rates for a class of iterative calculations, given a sufficient number of time steps and a cache that grows in size as a function of the memory balance and the calculation performed within the loop body. In this article, we give a generalization of time skewing for stencils on multiprocessor architectures, and discuss time skewing for multilevel caches.

## 1 Introduction

Time skewing [MW99, SSW97, Won99a] is a compile-time optimization that can achieve *scalable locality* for a class of iterative stencil calculations. That is, the cache hit rate can be made to grow with the problem size, allowing the use of extremely fast processors for extremely large problems. The full class of calculations for which time skewing can be used is defined in [Won99a]; for this article, we simply note that this class includes simple stencils in which each value is based only on values from the previous time step (such as the three point stencil calculation in Figure 1), and that more complex examples, such as the `TOMCATV` program of the SPEC95 benchmark set, can be coerced into this class.

Time skewing involves loop skewing and tiling, described by Wolf and Lam [WL91, Wol92], but can be applied to programs with imperfectly nested loops (such as Figure 1), which cannot be handled by their system. It can be viewed as the application of these techniques to the inter-iteration value-based flow dependences [PW93, PW98], followed by a transformation of the mapping of values to memory; or as a combination of loop alignment, imperfectly nested loop interchange, loop skewing, and tiling, optionally followed by a memory transformation.

In this article, we generalize time skewing for multiprocessor architectures. We begin by presenting the basic principles of our technique in terms of the simple example code shown in Figure 1: Section 2 reviews the uniprocessor time skewing transformation, and Section 3 introduces multiprocessor time skewing. Section 4 then discusses the application of this transformation to problems of higher dimensionality, and presents techniques for making use of multilevel caches. We then provide a more formal description of the general transformation in Section 5. Finally, we discuss related work in Section 6, outline plans for ongoing work in Section 7, and give our conclusions in Section 8.

## 2 Uniprocessor Time Skewing

We begin by producing a graph that describes the flow of values among the iterations of the program statements. As we are interested only in the flow of values, we eliminate any copies (such as the iterations of

*This is supported by NSF grant CCR-9808694

```
for (int t = 0; t<T; t++)
  {
  for (int i = 0; i<=N-1; i++)
    old[i] = 0.25 * (old[i-1] + old[i]+old[i] + old[i+1]);
  }
  for (int i = 1; i<=N-2; i++)
    cur[i] = 0.25 * (old[i-1] + old[i]+old[i] + old[i+1]);
  }
```

Figure 1: Three Point Stencil

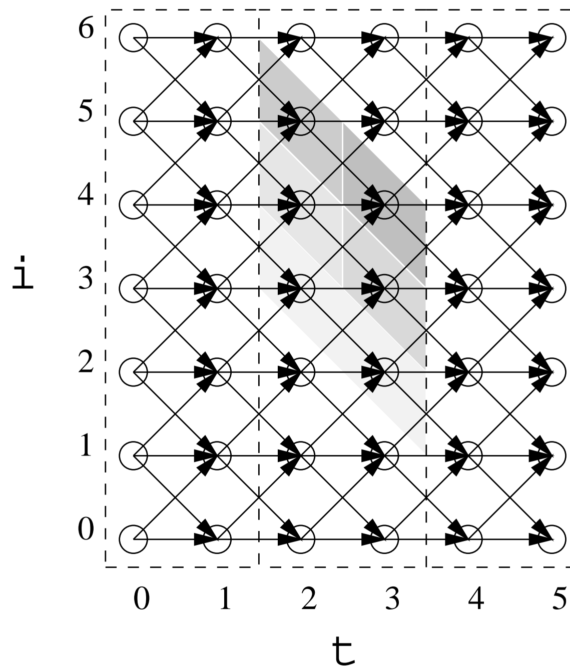


Figure 2: Time-Skewed Iteration Space for Three Point Stencil, for N=7, T=6

the first loop of Figure 1) by merging them with the uses of the copied values (this is like forward substitution, but we are modifying the value flow graph, not the program). Figure 2 shows the graph that would result from this analysis for Figure 1, if $N$ and $T$ were known to be 7 and 6. Note that the use of constants for $N$ and $T$ is only for illustration: We represent this graph using the Omega Library [KMP+95], which lets us represent such terms symbolically.

We then group the iterations into tiles (shown with dashed lines in Figure 2). The tiling ensures that the ratio of iterations within the tile to values flowing across the tile's boundaries (which we call the *compute balance*) grows with the width of the tile. In general, tile width would be significantly greater than that shown in the figure. Note that the compute balance gives the ratio of calculations to memory traffic arising from values that are live across tile boundaries. We can make this ratio arbitrarily large by increasing tile width, and thus can achieve arbitrarily high cache hit rate if we can (1) limit memory traffic for temporaries within a tile, and (2) prevent cache interference.

We next select an ordering for the iterations within each tile that ensures that the number of temporaries (i.e., values with lifetimes contained entirely within the tile) that are simultaneously live is independent of the problem size (though not the tile width). The shaded region of Figure 2 shows the progress of such an execution of the middle tile: The darkest iterations have been executed most recently.

Finally, we control memory traffic for temporaries, and cache interference, by creating an array as large as the number of simultaneously live temporaries, and storing all temporary values in this array. All memory traffic for iterations that are not at inter-tile boundaries will use this array: If it is smaller than the cache, it should remain entirely within the cache during the central iterations of the tile, and temporaries will not be written to memory, or interfere with each other. It is often possible to achieve high cache hit rates with simpler storage transformations – see [Won99b] for details.

To perform time skewing, we make extensive use of the integer tuple sets and relations provided by the Omega Library. We use them to represent iteration spaces, dataflow, the iteration space transformation, and the new mapping of values to memory locations. As we shall see in Section 5, this representation lets us give a concise definition of our transformation and then generate code with the Omega Library.

If we assume that floating point data and operations dominate the program, we can derive the tile size required for a given calculation (and thus, the amount of cache required) from the floating point parameters of the target architecture. Let $N$ be the number of iterations executed in each time step of the calculation, $O$ the number of floating point operations per iteration, $D$ the number of bytes of floating point data generated per iteration, $\tau$ the width of the tiles to be used in time skewing, $C$ the CPU speed of the target architecture (in MFLOPS), and $B$ its bandwidth to main memory (in Mbytes/sec). If we ignore the potential interference at the tile boundaries (this effect decreases with tile width), the main memory traffic for one tile is $2DN$ ($DN$ reads and $DN$ writes), which takes $\frac{2DN}{B}$ microseconds. To execute the calculations in $\tau$ time steps, the CPU requires $\frac{ON\tau}{C}$ microseconds. If $\tau \geq \frac{2DC}{OB}$, then $\frac{ON\tau}{C} \geq \frac{2DN}{B}$, and the CPU will have enough work to keep it busy during memory transfers. To keep three wavefronts in cache, we need $3D\tau$ bytes of cache memory.

For example, consider running Figure 1, for which $D = 8$ (one 8-byte value) and $O = 4$, on a machine with $C = 300$ and $B = 40$. If $\tau \geq 30$, memory traffic will take no more time than computation. Our "cache" array will require 720 bytes for $\tau = 30$, easily fitting in the L1 cache of any modern computer. In contrast, for large $N$, the data produced at the start of each time step will have been flushed from cache before the start of the next time step. Thus, each time step will require $\frac{2ND}{B}$ microseconds for memory access to load and store values for $N$ iterations, and $\frac{NO}{C}$ microseconds to perform its calculations; CPU utilization can be no more than $\frac{OB}{2DC}$ (about 3% for our hypothetical example).

Note that time skewing reduces memory bandwidth requirements; latency-hiding optimizations such as prefetching may still be needed to achieve full CPU utilization.

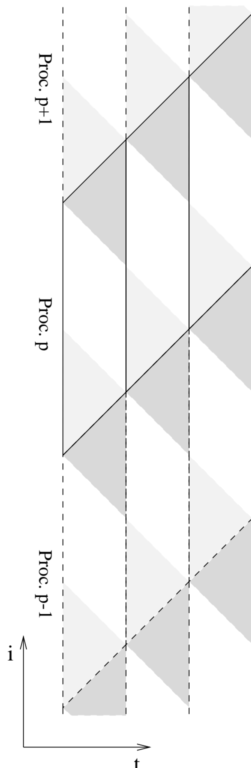


Figure 3: Blocking for Parallel Time Skewing (individual iterations not shown)

## 3 Multiprocessor Time Skewing

Although there are dependences among the tiles shown in Figure 2, we could achieve some level of concurrency by distributing these tiles across multiple processors and inserting posts and waits to ensure that the dependences are preserved. However, we can do much better by using the tiling shown in Figure 3 and hiding communication cost with an idea presented by Rosser in [Ros98, Chapter 7.1]: We divide each tile into three regions: the dark gray region at the top of the tile contains iterations that cannot be executed until data are received from the processor above; the light gray region at the bottom contains iterations that must be executed to produce results that are needed by the processor below; and the iterations in the white region in the center are not involved in communication. In Rosser's terminology, the light gray region is the "send slice", and the dark gray the "receive slice" (note that, due to the simple dependence patterns in the programs that we can optimize, we do not need to use Rosser's techniques to generate these slices).

Conveniently, the wavefront created by the time skewing algorithm executes the send slice first, then the central part of the tile, and then the receive slice. Thus, we can hide the cost of inter-processor communication by performing a send as soon as the send slice is completed, and delaying our wait for incoming data until after the central part of the tile is completed.

The boundary conditions for the first and last iterations are handled by enforcing whatever conditions are present in the original source code. For circular stencils, we treat the first and last processors as neighbors, with tiles identical to those of all other processors. For non-circular stencils, such as Figure 1, we rely on our code generation system to provide the right values for the edge iterations.

When we execute the sequence of blocks shown Figure 3 on $P$ processors, the non-edge tiles execute $\frac{N\tau}{P-1}$ iterations of the loop body, and the edge tiles execute no more than this many iterations. In the following

```
for (int t = 0; t<T; t++)
  {
  for (int i = 0; i<=N-1; i++)
    for (int j = 0; j<=N-1; j++)
      old[i][j] = cur[i][j];

  for (int i = 1; i<=N-2; i++)
    for (int j = 1; j<=N-2; j++)
      cur[i][j] = 0.125 *
                          (old[i-1][j] +
         old[i][j-1] + 4*old[i  ][j] + old[i][j+1] +
                           old[i+1][j]);
  }
```

Figure 4: Five Point Stencil

analysis, we focus on the non-edge tiles, as the edge tiles are smaller and should be completed faster. Let $\sigma = \frac{N}{P-1}$ (the size of the blocks of the i loop). Each tile loads $D\sigma$ bytes along its left edge, and stores the same number of bytes at the right edge (we will count the cost of accessing the values on the diagonal lines, which are communicated between processors, entirely in the communication cost below). As in the uniprocessor case, memory traffic will take no more time than computation if $\tau \geq \frac{2DC}{OB}$.

Each tile and sends (and receives) $2D\tau$ bytes. If the transfer of one block of data of this size takes less time than the execution of the white region in the center of the tile, then the entire communication cost will be hidden. This will be the case if $L + \frac{2D\tau}{B_N} \leq \frac{O}{C}(\sigma\tau - \tau^2)$, where $B_N$ is the network bandwidth (in Mbytes/sec) and $L$ is the network latency (in microseconds). Continuing the hypothetical example from the previous section, assuming $L = 1000$ and $B_N = 10$, and using the value $\tau = 30$, we can hide communication time if $\sigma > 2650$. If $L$ is small compared to $\frac{2D\tau}{B_N}$, we may be able to reduce the minimum required $\sigma$ by transferring the data is many small blocks, but we do not investigate that possibility here.

# 4 Multidimensional Arrays

Time skewing is somewhat more complicated for stencil calculations on multidimensional arrays, such as the code in Figure 4. If we were to block only the time step loop and traverse each tile with the diagonal wavefront $t + i + j$, the area of the wavefront would grow with $N$, and our "cache" array would not fit in cache for large problems. Thus, we block the time step loop and all but one of the inner loops, as shown (in simplified form) in Figure 5. This forces us to create additional arrays to store the values that are live across the diagonal block boundaries.

As for the one-dimensional case, we can derive the tile size (and thus the cache requirement) from the architecture parameters. Let $\sigma$ be the width of the tiles in the i loop. Each tile requires $\frac{ON}{C}\sigma\tau$ microseconds of CPU time, and $\frac{2DN}{B}(\sigma + 2\tau - 2)$ microseconds of memory access time for values that live across tile boundaries ($N\sigma$ are written at the end of the time block, and $2N(\tau - 1)$ at the end of the blocked i loop). Thus, we must ensure $\frac{\sigma\tau}{\sigma+2\tau-2} \geq \frac{2DC}{OB}$. The cache array requires $3D\sigma\tau$ bytes, and the tide array (which is not presumed to fit in cache), $2DN\tau$. For our hypothetical machine, we can set $\sigma = 78, \tau = 40$, producing a cache array of size 73K (note that $O = 6$ for Figure 4).

Note that our cache requirement grows with $(\frac{C}{B})^d$, where $d$ is the dimensionality of the array. Thus, when arrays of many dimensions are time skewed for machines with high $\frac{C}{B}$ (or *machine balance* [CCK88, McC95]), the cache array may not fit in the L1 cache. We will revisit this issue in Section 4.2.

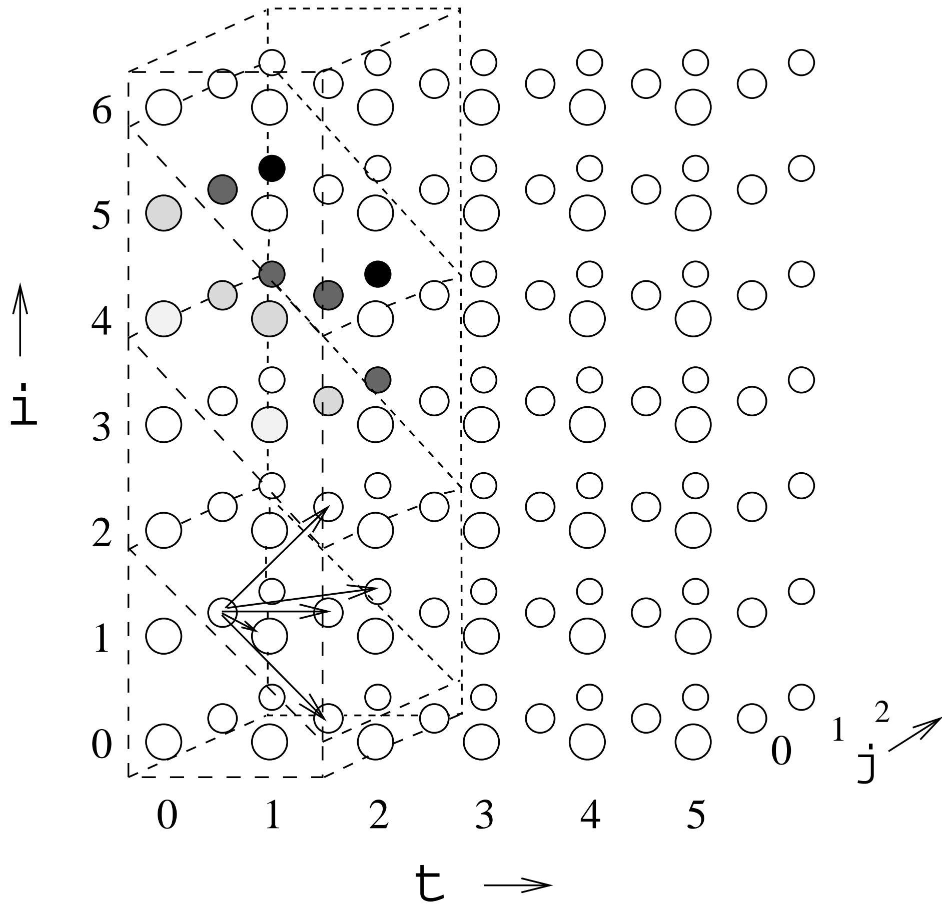


Figure 5: Time Skewed Iteration Space for Five Point Stencil

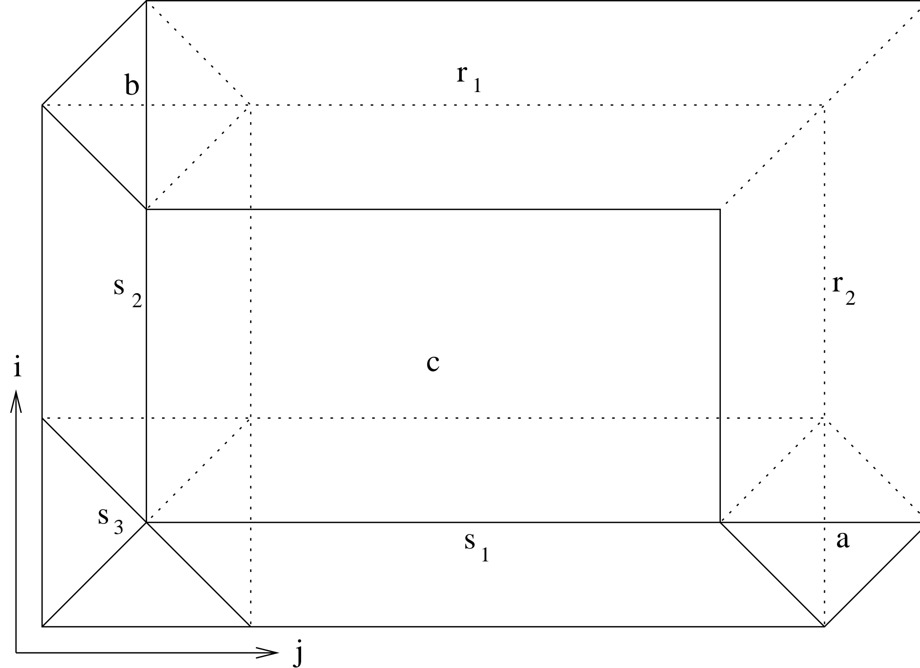


Figure 6: Single Tile for Parallel Time Skewing for 2-D Array, seen from $t = \infty$

## 4.1 Multiprocessors and Multidimensional Arrays

We can concurrently execute all the tiles shown for a given time block in Figure 5 in a manner similar to that shown for the one dimensional array: We first execute the triangular prism full of iterations that makes up the send slice, then the trapezoidal prism that makes up the center of the tile, and then the triangular prism containing the receive slice. Since the time skewed wavefront is skewed with respect to `j` as well as `i`, it does not automatically traverse these slices in order, so we must break the calculation up into these three phases. This introduces new memory traffic at the inter-slice boundaries (the increases in traffic is less than a factor of two). This strategy works for small-scale parallelism, but does not allow the use of $O(N^2)$ processors on the $N^2$ iterations: The width of the central slice shrinks with $\frac{N}{P-1}$, vanishing to nothing at $\frac{N}{P-1} = \tau$.

To achieve high degrees of parallelism, we must block both the `i` and `j` loops. Figure 6 shows one such tile, as seen from $t = \infty$. Note that this is a "head-on" view of a parallelepiped (a 3-d parallelogram), although the diagonal lines make it resemble the traditional sketch of a rectangular solid as seen from an angle. We use $\sigma_i$ and $\sigma_j$ to represent the widths of the blocks in the `i` and `j` loops. Without loss of generality, we assume $\sigma_i \leq \sigma_j$, as shown in the diagram. The send slice includes triangular prisms at the bottom ($s_1$) and left ($s_2$) edges of the tile (these intersect in the square pyramid $s_3$). The receive slice includes triangular prisms at the top ($r_1$) and right ($r_2$) edges (we have not found it necessary to label their intersection). The prisms of the receive slice have a face, rather than an edge, pointing at $t = \infty$. Tetrahedrons $a$ and $b$ are the intersections of $r$ and $s$. Region $c$, in the center of the tile, is in neither $r$ nor $s$.

The existence of regions $a$ and $b$ prevents us from ordering the calculation in three phases for $s$, $i$, and $r$. We can, however, use the following six phases:

1. Execute $s_3$ and send the data that will be needed for the $a$ and $b$ regions of neighboring tiles.

2. Execute the rest of $s_1$, performing a receive of the data required for $a$ only when it is needed, and then send the data that will be needed for the $r_1$ region of the neighboring tile.

3. Execute the rest of $s_2$, receiving and sending as above.
4. Execute $c$.
5. Receive data needed for $r_1$ and execute it.
6. Receive data needed for $r_2$ and execute it.

The ordering of iterations in each phase is exactly what is produced by the time-skewed wavefront – all iterations that share a given value of $t+i+j$ are executed before any iterations with a higher $t+i+j$. In some cases, this ordering leaves in cache exactly the values that will be needed in the next phase (as in the transition from $s_3$ to $s_1$). In other cases, the values that pass between phases must be stored until the next phase starts (as in the transition from $s_2$ to $c$). For each boundary along an edge of $s_1$, and $r_1$, we need to preserve approximately $2D\sigma_j\tau$ bytes, and for each edge of $s_2$ and $r_2$, $2D\sigma_i\tau$ bytes. If we are to hold all these boundary values in cache, we will need $8D\tau(\sigma_j+\sigma_i)$ bytes of cache, in addition to the $3D\tau\min(\sigma_j,\sigma_i)$ bytes for the "cache" array (we don't actually have to have storage for all boundaries simultaneously, but taking advantage of this complicates the analysis quite a bit and does not affect the largest of the boundary arrays, providing little benefit for the extra work).

Each tile executes $O\sigma_j\sigma_i\tau$ operations, and generates $2D\sigma_j\sigma_i$ bytes of main memory traffic (as in the case for one-dimensional arrays, we count the entire cost of moving data from one processor's cache to another processor's cache as part of the communication cost). As in the case of the single dimensional array, total memory access time for the tile will be less than CPU time if $\tau \geq \frac{2DC}{OB}$. This constraint is also sufficient to ensure that the send and receive slices (except $s_3$, which we consider too small to worry about) are each balanced on their own. For example, the region $s_1 - s_3 - a$ contains $\tau^2(\sigma_j - 2\tau)$ iterations, and performs $2\tau(\sigma_j - 2\tau)$ reads from main memory. We only need to ensure that $\tau \geq \frac{2DC}{OB}$ for it to require as much CPU time as memory access time.

To ensure that communication time is hidden, we must ensure (a) that the data sent at the completion of $s_3$ have arrived before the execution of $a$ begins, and (b) that the data sent at the completion of $s_1$ have arrived before the execution of $r_1$ begins. Each tile sends $2D\tau^2$ bytes from $s_3$ to the neighbor's $a$ phase, and then perform $O\tau^2(\sigma_j - 2\tau)$ operations before requiring data for its $a$ phase. Thus, we must ensure that $L + \frac{2D\tau^2}{B_N} \leq \frac{O}{C}\tau^2(\sigma_j - 2\tau)$. Each tile sends $2D\tau(\sigma_j - \tau)$ bytes from $s_1 - s_3$ to its neighbor's $r_1$ phase, and then executes regions $s_2 - s_3$ and $c$, which include just over $O(\sigma_j - \tau)(\sigma_i - 2\tau)$ operations, before requiring data for its $r_1$ phase. Thus, we must ensure that $L + \frac{2D}{B_N}\tau(\sigma_j - \tau) \leq \frac{O}{C}(\sigma_j - \tau)(\sigma_i - 2\tau)$.

When optimizing for our example architecture, our memory balance constraint requires that $\tau \geq 20$. For $\tau = 20$, communication constraint (a) simplifies to $\sigma_j \geq 245$; For $\tau = 20 \wedge \sigma_j = 245$, constraint (b) then becomes $\sigma_i \geq 1863$. This requires about 2.7M of cache: 115K for the "cache" array and 2635K for the boundary values.

## 4.2 Multi-level Tiling

Earlier work on time skewing did not cover multi-level tiling. If we produce a finer grained tiling using tiles like those of Figure 2, which span the entire range of values of `i`, we end up reducing performance rather than improving it. Each of the finer tiles still produces $DN$ values, which may not fit in any level of the cache; thus we have simply produced a narrower tile, with a lower compute balance. Fortunately, for one-dimensional arrays, we generally don't need to consider multi-level caches: The size of the "cache" array grows only linearly with the machine balance, and it can generally fit in the L1 cache.

The tiles produced by uniprocessor time skewing for multidimensional arrays are also unbounded in one dimension (for example, `j` in Figure 5). Thus, although we know the size of the "cache" array, we cannot put a static limit on the amount of data read from main memory by one such tile, making it impossible to ensure that smaller tiles of this shape will produce data that fits in a higher level of cache. This may limit the effectiveness of time skewing for two-dimensional problems on machines with high memory balance, though it is often possible to achieve good speedup by simply making $\tau$ and $\sigma$ as large as possible without making

“cache” spill out of the L1 cache. For arrays of higher dimension, the lack of multilevel tiling becomes an even bigger problem.

Our approach to tiling for uniprocessors with multilevel caches is simply to tile the last dimension, producing tiles of the same shape as the multiprocessor time skewing.

For example, suppose we are executing the code of Figure 4 on our hypothetical machine, but have only 32K of L1 cache and 2M of L2 cache, rather than the 115K of L1 cache that would be required to keep the CPU busy with the basic algorithm. As in the rest of this article, we will focus on bandwidth issues, assuming that we will either use other techniques to address latency or accept a certain number of misses at the beginning of each tile.

From the construction of the tile, we know that we must not generate any more main memory traffic if we are to keep the CPU busy. Thus, we must ensure that values that spill out of the L1 cache remain in the L2 cache, and produce an order for the iterations within the tile that does not exceed the available bandwidth to the L2 cache (which we will label $B_2$, and give the value 100 Mbytes/sec for our hypothetical machine).

Thus, we block the one remaining unblocked loop (for Figure 5, the skewed `j` loop) to create tiles for which (a) $\sigma_j$ is small enough for our “cache” array to fit in L1 and (b) $\sigma_j$ is large enough to keep L2 traffic within $B_2$. It is also necessary that (c) the array that holds the boundary values that pass between the new subtiles fits in L2. If we cannot satisfy all of these conditions, or if the entire set of working arrays exceeds the size of the L2 cache, we reduce some of the $\tau$ or $\sigma$ values, and settle for improving CPU utilization without maximizing it.

Returning to our example, condition (a) above requires that $3D\sigma_j\tau$ be less than the L1 cache size, or $\sigma_j \leq 34$; (b) requires that $\frac{O}{C}\sigma_j\sigma_i\tau \geq \frac{2D}{B_2}\sigma_i\tau$, or $\sigma_j \geq 8$ (c) requires that $2D\sigma_i\tau$ (49K in this case) fit in L2 (2M).

For the multiprocessor case, we start by assuming that all inter-phase boundary arrays will end up in L2 cache, as they are used infrequently compared to the “cache” array. If the “cache” array by itself will not fit in L1, we introduce a finer-grained blocking of the `j` loop within the $c$ region of the tile (and, if necessary, other regions). For our example architecture, the working arrays required more than the 2M of available L2 cache, so we would first have to reduce $\tau$ or $\sigma_i$ (giving up our goal of full processor utilization), and then produce sub-blocks to fit in L1. This arithmetic is left to the reader.

# 5 Describing the General Time Skewing Transformation

We make extensive use of the integer tuple sets and relations provided by the Omega Library in the definition and application of time skewing. This system provides a concise way to describe our iteration space transformation and the new mapping of values to memory locations, and provides features to automatically generate code from these descriptions.

## 5.1 Describing Iteration Space Transformations

The iteration space transformations we have presented can be described concisely in the framework of [KP94]. In this framework, each iteration of each statement is identified with a unique tuple of integers. These integers may correspond to the loop index value of a surrounding loop, or they may indicate which of several statements is being performed. For uniprocessors, we use the lexicographical order of these tuples to define the execution order of the statements and iterations. Thus, so we can describe an iteration reordering transformation as a mapping on the tuples assigned to the iterations. For multiprocessors, we identify which loops are distributed over processors (rather than time), and use the lexicographical order or the remaining variables to describe the execution order of the remaining iterations.

For example, the iteration spaces for the two statements shown in Figure 1 are

$$\{ \; [\, t, 1, i \,] \mid 0 \leq t < T \wedge 0 \leq i < N \; \}$$
$$\{ \; [\, t, 2, i \,] \mid 0 \leq t < T \wedge 1 \leq i < N-1 \; \}$$

After we merge statement 1 (the copy statement) into statement 2, only the second part of the iteration space remains. The iteration space reordering transformation that we use to produce the execution order shown in Figure 2 is

$$\{ \ [\ t, 2, i\ ] \rightarrow [\ t\ div\ \tau, i+t, t\ mod\ \tau\ ]\ \}$$

In this framework, loop skewing shows up in as sums of index variables ($t+i$), and tiling as *div* (for the tile number) and *mod* (for the offset within the tile).

Note that $\tau$ must be a known constant to manipulate these transformations with the Omega Library. We can turn the above descriptions of the original iteration space and the iteration space transformation into code that traverses the transformed iteration space using the code generation features of the Omega Library [KPR95].

To represent the blocking used in Figure 3, we map to a four-element tuple that represents the time block, the processor number, the wavefront within the processor, and the position on the wavefront:

$$\{ \ [\ t, 2, i\ ] \rightarrow [\ t\ div\ \tau, (t-i)\ div\ \sigma, i+t, t\ mod\ \tau\ ]\ \}$$

To generate the sends and receives, we could introduce tests into the program, to detect the start of wavefronts $i+t = 2\tau+1$ (when we can perform the send) and $i+t = \sigma - 2\tau$ (when me must perform the receive). Alternatively, we can map the iterations of the single original statement to three separate loop nests, for the send slice (nest 1 within the processor loop), center slice (nest 3), and receive slice (nest 5), and add send and receive statements (statements 2 and 4 within the processor loop). The iteration remapping (which does not contain the added statements) is

$$\begin{array}{ll} & \{\ [\ t,2,i\ ] \rightarrow [\ t\ div\ \tau, (t-i)\ div\ \sigma, 1, i+t, t\ mod\ \tau\ ] \mid i+t \leq 2\tau\ \} \\ \cup & \{\ [\ t,2,i\ ] \rightarrow [\ t\ div\ \tau, (t-i)\ div\ \sigma, 3, i+t, t\ mod\ \tau\ ] \mid 2\tau < i+t < \sigma - 2\tau\ \} \\ \cup & \{\ [\ t,2,i\ ] \rightarrow [\ t\ div\ \tau, (t-i)\ div\ \sigma, 5, i+t, t\ mod\ \tau\ ] \mid i+t \geq \sigma - 2\tau\ \} \end{array}$$

For stencils of larger dimension, the general iteration space transformation performed for uniprocessor time skewing is

$$\begin{array}{ll} \{\ [t, i_1, i_2 ... i_n] \rightarrow [ & t\ div\ \tau, \\ & (i_1+t)\ div\ \sigma_1, (i_2+t)\ div\ \sigma_2, ...(i_{n-1}+t)\ div\ \sigma_{n-1}, \\ & t + \Sigma_{l=1}^{n} i_l, \\ & (i_1+t)\ mod\ \sigma_1, (i_2+t)\ mod\ \sigma_2, ...(i_{n-1}+t)\ mod\ \sigma_{n-1}, \\ & t\ mod\ \tau\ ]\ \} \end{array}$$

Note that constant levels have been omitted in the interest of simplicity, as they do not arise for the examples shown in this paper. Furthermore, when a stencil includes values that are more than one element away, the angle of the skewing must be adjusted, as is the case whenever skewing is used [WL91], but we do not address that here. The above transformation also does not include the case in which multiple loop nests exist within the time step loop even after we have removed copy statements – for details of this case, see [Won99a, Won99b].

To produce the multiprocessor transformation, we must introduce blocks in the $i_n + t$ dimension, and separate out the different phases within the block:

$$\begin{array}{ll} \{\ [t, i_1, i_2 ... i_n] \rightarrow [ & t\ div\ \tau, \\ & (i_1+t)\ div\ \sigma_1, (i_2+t)\ div\ \sigma_2, ...(i_n+t)\ div\ \sigma_n, \\ & phase, \\ & t + \Sigma_{l=1}^{n} i_l, \\ & (i_1+t)\ mod\ \sigma_1, (i_2+t)\ mod\ \sigma_2, ...(i_n+t)\ mod\ \sigma_n, \\ & t\ mod\ \tau\ ] \\ \mid & \text{phase} - \text{constraints}\ \} \end{array}$$

where *phase* is a constant that controls the phase ordering, and the phase constraints identify the iterations for that phase. For example, for the tile shown in Figure 6, the phase constraints for each phase are:

| phase | constraints |
|---|---|
| $s_3$ | $bottom \wedge left$ |
| $a$ | $bottom \wedge right$ |
| $b$ | $top \wedge left$ |
| $c$ | $\neg(bottom \vee left \vee top \vee right)$ |
| $s_1$ | $bottom - left - right$ |
| $s_2$ | $left - bottom - top$ |
| $r_1$ | $top - left - right$ |
| $r_2$ | $right - bottom$ |

where

$$\begin{aligned} bottom &= \quad i + t \leq 2\tau \\ left &= \quad j + t \leq 2\tau \\ top &= \quad i + t \geq \sigma - 2\tau \\ right &= \quad j + t \geq \sigma - 2\tau \end{aligned}$$

## 5.2 Describing Memory Mappings

Integer tuple relations can also be used to represent the mapping from an iteration space to the index space of an array. For example, the storage mapping for the code in Figure 1 is

$$\{ \; [\; t, 1, i \;] \rightarrow \texttt{old}[i] \mid 0 \leq t < T \wedge 0 \leq i < N \; \}$$
$$\{ \; [\; t, 2, i \;] \rightarrow \texttt{cur}[i] \mid 0 \leq t < T \wedge 1 \leq i < N - 1 \; \}$$

The presence of conditions in the mapping allows us to use different storage mappings for different parts of the iteration space. For the transformed iteration space of Figure 2, we can store the values produced at the end of each time block in `cur` and the rest in `cache` with the mapping

$$\begin{aligned} & \{ \; [\; t, 2, i \;] \rightarrow \texttt{cur}[i] \mid t \; mod \; \tau = \tau - 1 \wedge 0 \leq t < T \wedge 0 \leq i < N \; \} \\ \cup \quad & \{ \; [\; t, 2, i \;] \rightarrow \texttt{cache}[(t + i) \; mod \; 3][t \; mod \; \tau] \mid 0 \leq t \; mod \; \tau < \tau - 1 \wedge 0 \leq t < T \wedge 0 \leq i < N \; \} \end{aligned}$$

We have given this description in terms of the original iteration space variables, rather than the transformed iteration space, because the conditions are somewhat clearer in this space. We can easily produce the mapping from the new iteration space to the storage locations by composing the above storage mapping with the inverse of the iteration space transformation. These storage mappings can also be used for automatic code generation, using the techniques given in [SW98].

The storage mapping used for general uniprocessor time skewing is

$$\begin{aligned} [t, i_1, i_2 ... i_n] \rightarrow \; & \texttt{cur}[i_1, i_2 ... i_n], && \text{when } tt = \tau - 1. \\ & \texttt{tide}_j[t + \Sigma_{l=1}^{n} i_l, xx_1, ... xx_{n-1}, tt], && \text{when } xx_j + 2 \geq \sigma_j \\ & && \wedge \; (\nexists k > j \text{ s.t. } xx_k + 2 \geq \sigma_k) \\ & && \wedge \; tt \neq \tau - 1, \\ & \texttt{cache}[(t + \Sigma_{l=1}^{n} i_l) \; mod \; 3, xx_1, ... xx_{n-1}, tt], && \text{otherwise,} \end{aligned}$$
where $xx_l = (i_l + t) \; mod \; \sigma_l$ and $tt = t \; mod \; \tau$

The `tide` arrays are used to hold the values that are live across blocks in the non-time-step loops. For stencils that read values that are more than one element distant, we must increase the constant 2 in the constraints that control which values are written to the `tide` arrays, but we do not discuss these details here.

The storage mappings required to provide separate arrays for the inter-phase boundaries can be produced in a straightforward (if somewhat tedious) application of the phase constraints to storage mapping above.

# 6 Related Work

Most current techniques for improving locality [GJ88, WL91, Wol92, MCT96] do not combine skewing and imperfectly nested loop interchange, and thus do not include time skewing. For time-step codes, these techniques are generally very successful in producing locality within the perfectly nested inner loops, but do not enhance locality between time steps. This generally places upper limits the cache hit rate that can be achieved, as the inner loops often have finite compute balance.

Pugh and Rosser [Ros98] optimize for locality by using *iteration space slicing* to find the set of calculations that are used in the production of a given element of an array. By ordering these calculations in terms of the final array element produced, they achieve an effect that is similar to a combination of loop alignment and fusion. However, their system transforms the body of the time loop, without reordering the iterations of the time loop itself, and is thus also limited by the finite balance of the calculation in the loop body.

Work on tolerating memory latency, such as that by [MLG92], complements work on bandwidth issues. Optimizations to hide latency cannot compensate for inadequate memory bandwidth, and bandwidth optimizations do not eliminate problems of latency. However, we see no reason why latency hiding optimizations cannot be used successfully in combination with time skewing.

Recent work by Song and Li [SL99] also uses a combination of loop skewing and tiling, and limited array expansion. This system performs an iteration space transformation that is similar to uniprocessor time skewing, except that the time step loop, rather than one of the spatial loops, is unbounded. Their use of array expansion produces slightly higher memory traffic than our system, but the effect of this traffic on bandwidth can be offset by selecting a slightly larger block size. However, Song and Li only discuss optimizations for uniprocessors.

# 7 Future Directions

The time skewing transformations described in [MW99] and this article can be used to achieve scalable locality for some calculations on either uniprocessor or multiprocessor machines, even when the techniques of Wolf and Lam are not applicable. However, the applicability of time skewing has not yet been tested over a wide range of benchmarks. It may be widely applicable in its current form; it may be applicable only after additional preprocessing steps have been added to those given in [Won99a]; or it may not be widely applicable, either because the technique is too limited or something in most codes fundamentally inhibits scalable locality. A study of benchmark codes could do much to resolve these issues.

We can detect the potential for scalable locality by looking at the overall compute balance of the entire program. Specifically, we symbolically compare the total number of calculations (for example, $N^3$ for a program that performs a single matrix multiply, or $N^2T$ for a program consisting only of Figure 1) to the number of non-temporary values for the program as a whole (the total I/O of the program – $3N^2$ for the matrix multiply program or $N^2$ for the Figure 1 program). If this balance does not scale with the problem size, it represents a fundamental limit on locality. On the other hard, if the overall program has scalable compute balance, then our ability to produce scalable locality depends on our ability to reorder the calculations and/or remap storage to keep temporaries from being stored in main memory.

Overall balance and the opportunity to perform time skewing can both be checked automatically. If many codes with scalable reuse cannot be time skewed, then manual examination of these codes may be needed to determine whether simple preprocessing for time skewing, or a more fundamental modification of the technique, may enable scalable locality.

We have already demonstrated the effectiveness of uniprocessor time skewing at compensating for extremely high memory balance [SSW97, Won99b]. We hope to produce a similar demonstration for multiprocessor time skewing, using a cluster of workstations, once this cluster has been installed later this summer.

## 8 Conclusions

Time skewing can be used to transform iterative stencil calculations into a form that allows the efficient use of computers with arbitrarily high main memory machine balance, given an L1 cache that grows in size as a function of the balance. We have shown that this transformation can be generalized to allow the use of multiprocessor computers with arbitrarily high memory balance and arbitrarily low network performance, (once again, given sufficient L1 cache).

We have also shown how to derive tile size, and thus L1 cache requirement, from the program being optimized and the performance parameters of the target architecture. When the target machine does not have sufficient cache, we can (a) produce smaller tiles that fit in L1 but create stalls waiting for main memory, (b) use tiles that fit in L2, possibly creating stalls waiting for L2, or (c) use multi-level tiling to prevent stalls altogether, given sufficient bandwidth to L2.

## References


[CCK88] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358, August 1988.

[GJ88] D. Gannon and W. Jalby. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, pages 587–616, 1988.

[KMP+95] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, Dept. of Computer Science, University of Maryland, College Park, March 1995. The Omega library is available from http://www.cs.umd.edu/projects/omega.

[KP94] Wayne Kelly and William Pugh. Determining schedules based on performance estimation. *Parallel Processing Letters*, 4(3):205–219, September 1994.

[KPR95] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *The 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, McLean, Virginia, February 1995.

[McC95] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Technical Committee on Computer Architecture Newsletter*, Dec 1995.

[MCT96] K.S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Trans. on Programming Languages and Systems*, 18(4):424–453, 1996.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[MW99] John McCalpin and David Wonnacott. Time skewing: A value-based approach to optimizing for memory locality. Technical Report DCS-TR-379, Dept. of Computer Science, Rutgers U., February 1999. Available as ftp://www.cs.rutgers.edu/pub/technical-reports/dcs-tr-379.ps.Z.

[PW93] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the 6th Annual Workshop on Programming Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, August 1993. Also available as Tech. Report CS-TR-3196, Dept. of Computer Science, University of Maryland, College Park.

[PW98] William Pugh and David Wonnacott. Constraint-based array dependence analysis. *ACM Trans. on Programming Languages and Systems*, 20(3):635–678, May 1998. http://www.acm.org/pubs/citations/journals/toplas/1998-20-3/p635-pugh/.

[Ros98] Evan J. Rosser. *Fine-Grained Analysis of Array Computations.* PhD thesis, Dept. of Computer Science, The University of Maryland, September 1998.

[SL99] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. In *ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, May 1999. To appear.

[SSW97] Tina Shen, Jaime Spacco, and David Wonnacott. High MFLOP rates for out of core stencil calculations using time skewing. In *SC '97 poster session*, November 1997. Available as http://www.haverford.edu/cmsc/davew/cache-opt/SC97poster.ps.

[SW98] Tina Shen and David Wonnacott. Code generation for memory mappings. In *The 1998 Mid-Atlantic Student Workshop on Programming Languages and Systems (MASPLAS '98)*, April 1998. An updated version is available as http://www.haverford.edu/cmsc/davew/cache-opt/mmap.ps.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 1991.

[Wol92] Michael Edward Wolf. *Improving Locality and Parallelism in Nested Loops.* PhD thesis, Dept. of Computer Science, Stanford U., August 1992.

[Won99a] David Wonnacott. Achieving scalable locality with time skewing. Technical Report DCS-TR-378, Dept. of Computer Science, Rutgers U., February 1999. Available as ftp://www.cs.rutgers.edu/pub/technical-reports/dcs-tr-378.ps.Z.

[Won99b] David Wonnacott. Time skewing: A value-based approach to optimizing for memory locality. In preparation. A preprint is available as http://www.haverford.edu/cmsc/davew/cache-opt/tskew.ps, and parts of this work are included in Rutgers University CS Tech Reports 379 and 378., February 1999.