Extended version for IPDPS 2000 electronic publication (with additional material in Section 7)

Using Time Skewing to Eliminate Idle Time due to Memory Bandwidth and Network Limitations

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

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 sufficient cache memory. Thus, it can eliminate processor idle time caused by inadequate main memory bandwidth.

In this article, we give a generalization of time skewing for multiprocessor architectures, and discuss time skewing for multilevel caches. Our generalization for multiprocessors lets us eliminate processor idle time caused by any combination of inadequate main memory bandwidth, limited network bandwidth, and high network latency, given a sufficiently large problem and sufficient cache. As in the uniprocessor case, the cache requirement grows with the machine balance rather than the problem size. Our techniques for using multilevel caches reduce the L1 cache requirement, which would otherwise be unacceptably high for some architectures when using arrays of high dimension.

1 Introduction

Time skewing [8, 15, 21] is a compile-time optimization that can achieve *scalable locality* for a class of iterative stencil calculations, given a sufficient number of time steps and sufficient cache memory. That is, the cache hit rate can be made to grow with the problem size, and we can eliminate (for large problems) idle time due to high *machine balance* (the ratio of processor speed to memory bandwidth [1, 9]). Scalable locality lets us apply processors with increasing machine balance to increasingly large problems, just as scalable parallelism lets us apply higher levels of parallelism to larger problems. The cache required for time skewing grows with the machine balance, but is independent of (or, in some cases, grows less than linearly with) the problem size (a high degree of locality generally occurs automatically if all program data fits in cache).

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

Figure 1. Three Point Stencil

The full class of calculations for which time skewing can be used is defined in [21]. For this article, we simply note that we work with code in which each value can be expressed as a function of values from the previous time step (such as the three point stencil calculation in Figure 1), and allow procedures with imperfectly nested loops (such as Figure 1 and the `TOMCATV` program from SPEC95).

Time skewing involves loop skewing and tiling, described by Wolf and Lam [20, 19], but is more general, as it can be applied to programs with imperfectly nested loops. It can be viewed as the application of these techniques to the inter-iteration value-based flow dependences [13], 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 and architectures with multilevel caches. Section 2 reviews the uniprocessor time skewing transformation in terms of the simple example code shown in Figure 1. Section 3 introduces multiprocessor time skewing (again, using Figure 1), and gives formulas for the amount of cache memory and amount of work per processor that are needed to fully eliminate processor idle time. Section 4 discusses uniprocessor time skewing for problems of higher dimensions, and Section 5 extends our multiprocessor algorithm (and formulas) to this domain. Section 6

presents our techniques for making use of multilevel caches. Section 7 spells out the details of the general time skewing transformation. We discuss related work in Section 8 and give our conclusions in Section 9.

2 Uniprocessor Time Skewing

Our exposition of the uniprocessor time skewing algorithm will follow its application to the code in Figure 1. We begin by producing a graph that describes the flow of values among the iterations of the program statements. The left side of Figure 2 shows the graph that would result from this analysis, for $N = 9$ and $T = 6$. Note that the use of constants for N and T is only for illustration: We represent this graph using the Omega Library [5], which lets us represent such terms symbolically. In this figure, the squares correspond to executions of the copy statement, and circles to executions of the computation; solid arrows indicate data flow, and dashed arrows other dependences.

We can perform time skewing on this iteration space, or on the program that results if we perform array expansion and copy propagation (on the arrays), as shown on the right of Figure 2. We will focus on the latter in our discussion, as it requires less cache space (by about a factor of 2) and requires a somewhat simpler iteration space transformation (as the loops are perfectly nested), though it does require our storage remapping techniques [16].

We first identify the most frequently executed region(s) of the code. We can either approximate this by finding the code that is surrounded by the greatest number of loops with non-constant upper bounds, or produce a more accurate answer via symbolic volume computation [12]. In either case, we perform this analysis interprocedurally (we need a DAG showing all possible calls, so our analysis does not handle recursive procedures). The two statements of Figure 1 are equally frequent.

For each computationally intensive region, we measure the *compute balance* (the ratio of floating point calculations within a region to floating point values that are live across the region's boundaries) of the surrounding loops, until we find a scope at which the compute balance grows as a factor of the input. We say that such code exhibits *scalable compute balance*. If no loop level exhibits scalable balance, it is not possible to produce scalable locality for the program. Neither inner loop of Figure 1 has scalable balance, but the outer loop does (it performs $O(NT)$ operations on $O(N)$ live values).

We then group the iterations of these loops into tiles (shown with dashed lines in Figure 2). The tiling we use ensures that each tile also has scalable compute balance (in this case, the 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 regions in Figure 2 shows the progress of such an execution of the middle tile: The darkest iterations have been executed most recently. Note that, at any one time, the values that are live but will be consumed within the tile are all contained in the most recent wavefronts.

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. We use an array that holds three wavefronts, though it is possible to use an array that is one element larger than two wavefronts (at the cost of more complicated code). 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 [22] 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 7, this representation lets us give a concise definition of our transformation and 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, τ 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 τ 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

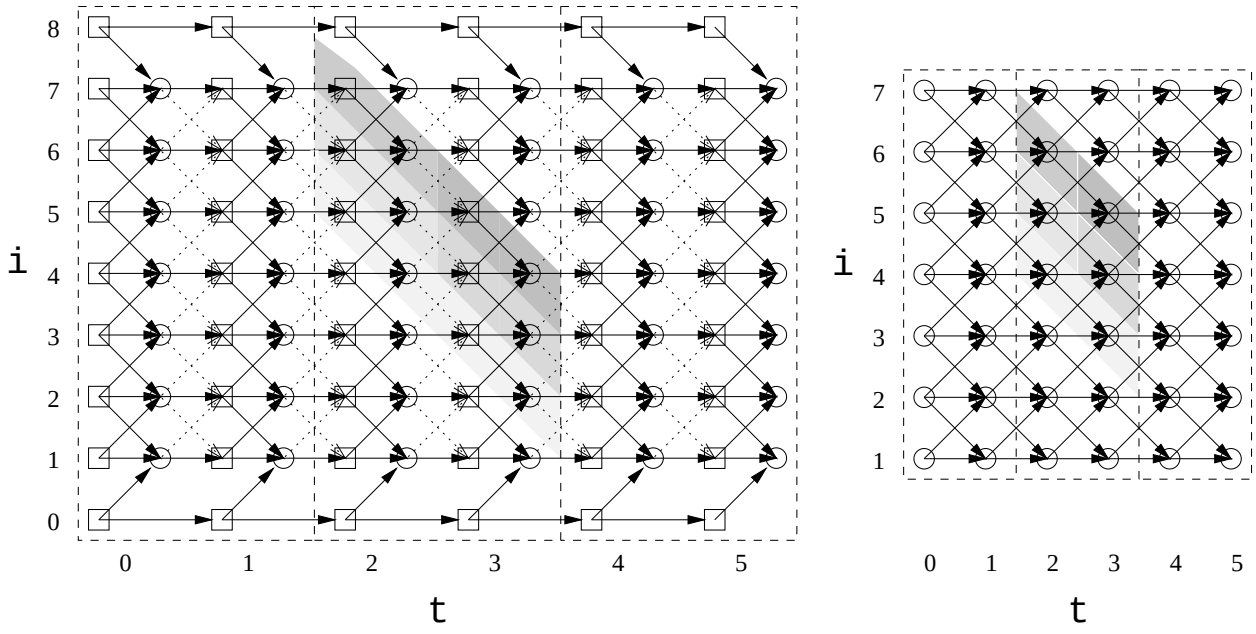

Figure 2. Time-Skewed Iteration Spaces (Fig. 1, N=9, T=6), Without & With Copy Prop.

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, when the original code is run with 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.

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 [14, 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" (when optimizing a program with a simple dependence patterns, like Figure 1, 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.

Unlike conventional ("atomic") tiles, our "molecular tiles" [1] do not correspond to a single perfect loop nest. We must insert sends and receives, either by generating three separate loop nests (for the three "atoms" of each molecule) with communication in between, or by generating communication statements guarded by conditionals.

When we execute the sequence of tiles 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 analysis, we focus on the non-edge tiles, as the edge tiles are smaller and should

[1] Thanks to Sanjay Rajopadhye for suggesting this term.

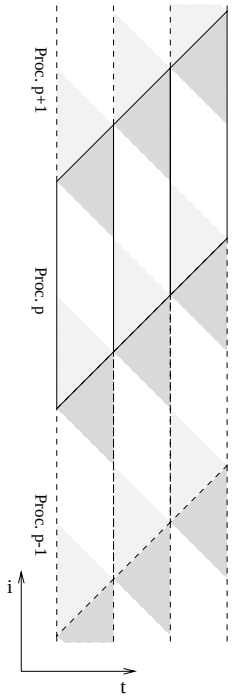

Figure 3. Blocking for Parallel Time Skewing (individual iterations not shown)

be completed faster.

Let $\sigma = \frac{N}{P-1}$ (the size of the blocks of the `i` loop). Each tile loads $D(\sigma + 2\tau)$ bytes along its left and top edges, stores the same number of bytes at the right edge, Thus, memory traffic will take no more time than computation if $\frac{O\tau\sigma}{C} \geq \frac{2D(2\tau+\sigma)}{B}$. Each tile 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). If L is small compared to $\frac{2D\tau}{B_N}$, we may be able to reduce the minimum required σ by transferring the data is many small blocks, but we do not investigate that possibility here.

We can minimize the cache requirement (as in the uniprocessor case, $3D\tau$) by minimizing τ, which can be made almost as small as $\frac{2DC}{OB}$ (though σ becomes very large as τ approaches this limit). Alternatively, we can reduce σ (and gain more parallelism for a given problem size) by increasing τ and using more cache.

Continuing the hypothetical example from the previous section, assume $L = 1000$ and $B_N = 10$. Our constraints are (a) to match compute balance to machine balance, $160\tau\sigma \geq 4800\tau + 4800\sigma$, and (b) to hide communication time, $40\tau\sigma \geq 40\tau^2 + 4800\tau + 3000000$. We can reduce our cache to approximately what was needed in the uniprocessor code by making τ close to 30. If we set $\tau = 30$, constraint (b) would require $\sigma > 2650$, but we could not satisfy constraint (a) (though we would be very close to the proper compute balance, given $\sigma > 2650$). The values $\tau = 31, \sigma = 2571$ satisfy both constraints, as do $\tau = 60, \sigma = 1430$ (which uses 1440 bytes of cache) and $\tau = 682, \sigma = 912$ (which uses 16K bytes of cache). Thus, if our hypothetical machine has 16K of L1 cache, we can eliminate idle time due to both memory access and communication when $N > 912P$.

Handling more complex codes raises a number of other issues. The boundary conditions for the first and last iterations are handled by enforcing whatever conditions are present in the original source code. For non-circular stencils, such as Figure 1, we rely on our code generation system to provide the right values for the edge iterations. For circular stencils, we treat the first and last processors as neighbors, with tiles identical to those of all other processors. We can generate the iteration space transformation for more complex cases (such as those with more complicated sets of imperfectly nested loops) using the iteration Rosser's "iteration space slicing" techniques [14]: We try several groupings of the set of values that is live after a block of time steps, slicing backward to find the set of iterations that must be executed to produce a given group. If this produces tiles with scalable balance, we continue with the time skewing algorithm; if not, we cannot optimize the code. We also have a less general algorithm to find such tilings that does not require the techniques of [14] (this algorithm is general enough to handle the `TOMCATV` benchmark code). The introduction of multiple processors does not complicate these issues, so the details are left in our work on uniprocessor time skewing [8, 21].

4 Multidimensional Arrays

For programs with multidimensional arrays, such as the code in Figure 4, we must do more than just block the time loop and adjust the wavefront. 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. As in Figure 2, the solid arrows show dataflow (for simplicity, not all arcs are shown), dashed lines enclose tiles (only the first time block is shown), and shading shows the progress of the wavefront within a tile.

Tiling the extra loops forces us to create additional arrays, which we call "tide" arrays, to store the values that are live across the diagonal tile boundaries.

```
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

As in the one-dimensional case, we can derive the tile size (and thus the cache requirement) from the architecture parameters. Let ρ be the width of the tiles in the `i` loop. Each tile requires $\frac{ON}{C}\rho\tau$ microseconds of CPU time, and $\frac{2DN}{B}(\rho+2\tau)$ microseconds of memory access time for values that live across tile boundaries ($N\rho$ are written at the end of the time block, and $2N\tau$ at the end of the blocked `i` loop). Thus, we must ensure $\frac{\rho\tau}{\rho+2\tau} \geq \frac{2DC}{OB}$. The cache array requires $3D\rho\tau$ bytes, and the tide array (which is not presumed to fit in cache), $2DN\tau$. To minimize cache, we minimize the value of $\rho\tau$ under the above inequality. The value of $\rho\tau$ as a function of ρ is $\frac{2\Gamma\rho^2}{\rho-4\Gamma}$, where we have combined various machine parameters as $\Gamma = \frac{DC}{OB}$. We set the derivative of this function [18, Chapter 10], $\frac{(\rho-4\Gamma)(4\Gamma\rho)-(2\Gamma\rho^2)(1)}{(\rho-4\Gamma)^2}$, to zero, and solve to get $\rho = 8\Gamma$, which gives us $\tau = 4\Gamma$. For our hypothetical machine, this gives $\rho = 80, \tau = 40$, producing a cache array of size 75K (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 $\frac{C}{B}$ (machine balance), the cache array may not fit in the L1 cache. We will revisit this issue in Section 6.

5 Multiprocessors and Multidimensional Arrays

For multiprocessors, we can distribute the tiles of Figure 5 across processors – essentially extending the tiles of Figure 3 in the `j` dimension. This produces triangular prisms for the send and receive slices, and a parallelogram prism for the center of the tile. 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 generate the code for our molecular tiles as three separate loop nests. Furthermore, the final iterations of each subtile do not leave in cache the values needed at the start of the next subtile, introducing new memory traffic at the inter-subtile boundaries (the increase in traffic is less than a factor of two). The send and receive slices may not, by themselves, be balanced, and since they scale in size with N, we cannot in general use prefetching to offset this problem. We must therefore "widen" the send and receive slices into trapezoidal (rather than triangular) prisms, to achieve the proper balance in each of the three subtiles. 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 tile in the tiling we use, 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. The thick lines give the outline of the tile, and the thin lines the subtiles. If we were to extend the subtiles all the way across the `j` loop (and ignore the shading), this would be the aforementioned tiling for limited parallelism: the trapezoidal prism at the bottom is the send slice, and the trapezoidal prism at the top the receive slice.

For $O(N^2)$ parallelism, the tiles (in general) have neighbors to the north (increasing `i`), south (decreasing `i`), east (increasing `j`), and west (decreasing `j`), as well as in the time dimension. We therefore find send and receive slices for each of the three subtiles, and extend these regions somewhat to include all iterations behind (or ahead of) any wavefront that includes an element in the send (or receive) slice. Figure 6 shows the send (light gray) and receive (dark gray) regions only for the southern subtile. This allows us to execute the iterations of each tile in the following order:

1. Execute the send region in the southern subtile and send data across boundary s_1.
2. Execute the central region of the southern subtile.
3. Receive the data from boundary r_2 (sent by the eastern neighbor in Step 1), finish executing the southern subtile, and send data across boundary s_3.
4. Execute the central subtile (between the southern and northern ones), performing send s_4 and receive r_5 as necessary.
5. Receive data across boundary r_6, and execute the northern subtile, performing send s_7 and receive r_8 as necessary.

Within each region, we use the wavefront that was used by the uniprocessor time skewing algorithm – all iterations that share a given value of $t+i+j$ are executed before any iterations with a higher $t+i+j$. From the construction of the send and receive regions of each subtile, we know that the data that were in the "cache" array at the end of one region

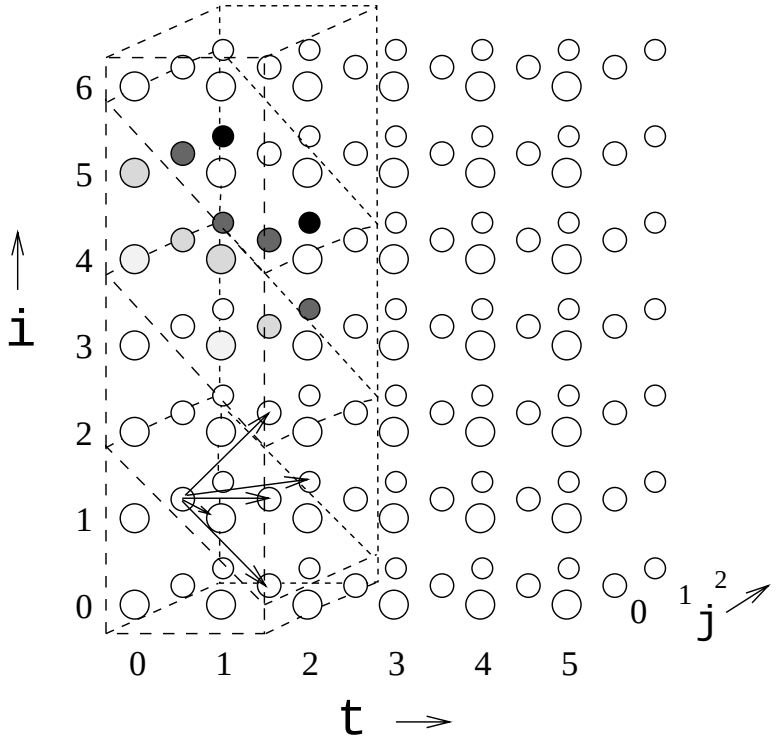

Figure 5. Time Skewed Iteration Space for Five Point Stencil

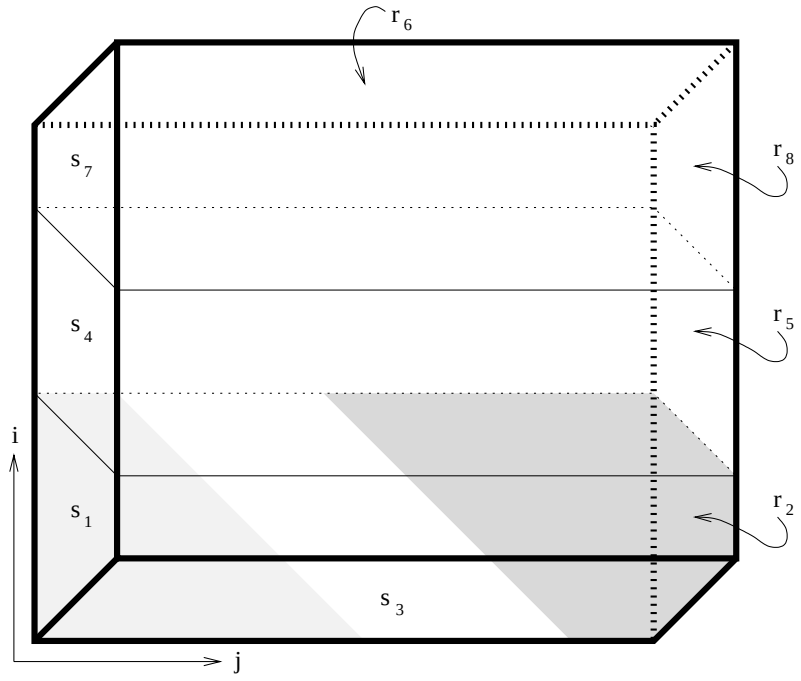

Figure 6. Single Tile for Parallel Time Skewing for 2-D Array, seen from $t = \infty$

will be exactly what is needed at the start of the next region. However, data may not remain in cache between the three subtiles. Thus, we must ensure that the compute balance of each subtile (rather than the overall balance of the tile) is sufficient to keep the processor busy, and that each tile is large enough to hide communication across the east/west and north/south boundaries. In some cases, we may need to use several central subtiles, (stacked in the `i` dimension) to hide the north/south communication.

Let τ be the width of the tiles in the time dimension, σ be the width of the tile in the `j` dimension, and k be the number of central subtiles used (one, in Figure 6). We require that all subtiles have equal cross-sectional area (in the `i-t` dimensions), and let μ be this area divided by τ (that is, height of the central subtile). We can ensure that neither memory access time nor communication time exceeds processor time by satisfying the following constraints:

1. $\mu \geq \tau$ (This follows from the construction of the north and south subtiles.)

2. $\frac{O}{C}\mu\tau\sigma \geq \frac{D}{B}(2\mu\sigma + 4\tau\sigma + 4\mu\tau)$ (The CPU time for each subtile must be $\geq$ its memory access time.)

3. $\frac{O}{C}\mu\tau(\sigma - \mu) \geq L + \frac{2D}{B}\mu\tau$ (The CPU time for the central region of each subtile must be $\geq$ the east/west communication time.)

4. $\frac{O}{C}k\mu\tau\sigma \geq L + \frac{2D}{B}\tau\sigma$ (The CPU time for the k middle subtiles must be $\geq$ the north/south communication time.)

The cache requirement will be $3D\,\mu\tau$. Note that the minimum $\mu\tau$ for Constraint 2 above is achieved when $\sigma \to \infty$, that Constraint 3 can be satisfied for any μ and τ by increasing σ, and that Constraint 4 can be satisfied by increasing k. To find the minimum cache requirement, we therefore minimize $\mu\tau$ for $\mu \geq \tau \wedge \frac{O}{C}\mu\tau \geq \frac{2D}{B}\mu + \frac{4D}{B}\tau$. Note that this is the same constraint we used for the uniprocessor time skewing (with μ substituted for ρ), and once again it is minimized for $\mu = 2\tau = \frac{4DC}{OB}$. Given μ and τ, we can easily solve Constraints 2 and 3 for σ and Constraint 4 for k.

Once again, we may come close to the uniprocessor cache requirement if we are willing to accept a large σ, or reduce σ at the cost of more cache. Interesting values for our hypothetical architecture include $\tau = 40, \mu = 81, \sigma = 6481, k = 1$ (which requires 76K of cache) and $\tau = 40, \mu = 136, \sigma = 195, k = 1$ (which requires 128K of cache). Each processor will run τ time steps on $k\mu\sigma$ array elements. For a q by q array of processors, the above tilings require problems of size $\frac{N}{q} \geq \sqrt{k\mu\sigma}$, or $N \geq 725q$ and $N \geq 162q$ respectively.

6 Multi-level Tiling

Earlier work on time skewing did not cover multi-level tiling. If we produce a finer grained tiles of the shape shown in 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 one level of cache and high memory balance, though it is often possible to achieve good speedup by simply making τ and ρ 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 block the last dimension, producing tiles of the same shape as the multiprocessor time skewing. This lets us further reduce the size of the "cache" array without introducing any new main memory traffic, as long the L2 cache can hold the values needed across the boundaries introduced by this last level of tiling.

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 unblocked loop (for Figure 5, the skewed `j` loop) to create tiles for which

1. $3D\sigma\tau \leq$ L1 cache size (σ is small enough for our "cache" array to fit in L1.)

2. $3D\sigma\tau + 4D\rho\tau + 2D(\sigma\rho + 2\sigma\tau) \leq$ L2 cache size (The L2 cache must hold everything in the L1 cache (assuming a write-through cache), the arrays that carry values across the `j` tile boundaries, and everything that is written to main memory during the execution of one tile.)

3. $\frac{4D\rho\tau+2D(\sigma\rho+2\sigma\tau)}{B_2} \leq \frac{O}{C}\sigma\rho\tau$, (Memory access time for the L2 cache (for values across the `j` tile boundaries and to main memory) does not exceed CPU time.)

In the above, we have assumed that the cache system will (a) keep the "cache" array (containing values on the wavefront of execution within the tile) in L1 cache, and (b) keep values that are live between the tiles in the `j` dimension in L2 cache, and thus write only other values to main memory. We believe that (a) is justified by the fact that most of our "cache" array is accessed in each wavefront, and that (b) is justified by the fact that Condition 2 lets us keep the entire "output" of one tile in L2. Architectures that allow the compiler to control the cache system would remove the need for these assumptions, and let us remove the term $2D(\sigma\rho + 2\sigma\tau)$ (the traffic we want to direct to main memory) from Condition 2. If we cannot satisfy the three conditions above, we reduce the value of ρ, and settle for improving CPU utilization without maximizing it.

Returning to our example, suppose our hypothetical uniprocessor machine has only 32K of L1 cache and 2M of L2 cache, rather than the 76K of L1 cache that would be required to keep the CPU busy when running the code of Figure 4 with the basic algorithm. Condition 1 requires that $\sigma \leq 34$; Condition 2 requires that $\sigma \leq 516$; Condition 3 requires that $\sigma \geq 27$. We can therefore achieve full CPU utilization with any σ between 27 and 34.

For the multiprocessor case, we use the value of σ we derived in Section 5, which we will now call σ_p, to distribute iterations to processors. We derive the block size that is needed for the available L1 and L2 caches as we did in the uniprocessor case, using μ instead of ρ (we call this σ_m). On each processor, we block the skewed `j` loop into blocks of size σ_m. Note that $\sigma_m < \sigma_p$ (unless L2 is smaller than L1), and $\sigma_m \geq \tau$. In general, σ_m may be less than μ, and the first extended send and receive regions (the shaded regions in Figure 6) will span several blocks. In this case, we simply do not extend these regions across blocks (we did so originally only to improve locality). This also allows us to modify the constraints on the size of the tiles, yielding slightly smaller tiles.

7 Generalized Time Skewing

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 can automatically generate code from these descriptions.

7.1 Describing Iteration Space Transformations

The iteration space transformations we have presented can be described concisely in the framework of [6]. 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 perform array expansion and copy propagate statement 1 (the copy) 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 on the right side of Figure 2 is

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

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 τ must be a known constant to manipulate these transformations with the Omega Library. We can use the code generation features of the Omega Library [7] to turn the above descriptions of the original iteration space and the iteration space transformation into code that traverses the transformed iteration space.

To represent the tiling 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

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

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

$$\begin{aligned}
\{\ [t, i_1, i_2 ... i_n] \rightarrow [\ & t\ div\ \tau, \\
& (t+i_1)\ div\ \rho_1, (t+i_2)\ div\ \rho_2, \\
& ...(t+i_{n-1})\ div\ \rho_{n-1}, \\
& t + \Sigma_{l=1}^{n} i_l, \\
& (t+i_1)\ mod\ \rho_1, (t+i_2)\ mod\ \rho_2, \\
& ...(t+i_{n-1})\ mod\ \rho_{n-1}, \\
& t\ mod\ \tau\ ]\ \}
\end{aligned}$$

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 [20], 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 [21, 22].

To produce the multiprocessor transformation, we must define the tile borders with $t-i$ rather than $t+i$, block every dimension, and separate out the subtiles and regions within a subtile:

$$\begin{aligned}
\{\ [t, i_1, i_2 ... i_n] \rightarrow [\ & t\ div\ \tau, \\
& (t-i_1)\ div\ \rho_1, (t-i_2)\ div\ \rho_2, \\
& ...(t-i_{n-1})\ div\ \rho_{n-1}, (t-i_n)\ div\ \rho_n, \\
& subtile, \\
& region, \\
& t + \Sigma_{l=1}^{n} i_l, \\
& (t+i_1)\ mod\ \rho_1, (t+i_2)\ mod\ \rho_2, \\
& ...(t+i_{n-1})\ mod\ \rho_{n-1}, (t-i_n)\ mod\ \rho_n, \\
& t\ mod\ \tau\ ] \\
\mid\ & \text{subtile and region constraints}\ \}
\end{aligned}$$

where $subtile$ is a constant that identifies which of the subtiles we are describing (north, center, or south), and $region$ is a constant that identifies the region within the subtiles. As for the one-dimensional array, we will insert send and receive statements between the subtiles and regions.

If we wish to tile for multi-level caches, as in Section 6, we introduce a level $(t+i_n)\ mod\ \sigma_m$ just above the level that gives the wavefront ($t+\Sigma_{l=1}^{n} i_l$). For the multiprocessor version, this also affects the region constraints, as discussed in Section 6.

7.2 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

$$\begin{aligned}
&\{\ [t,1,i\ ] \rightarrow \texttt{old}[i] \mid 0 \leq t < T \wedge 0 \leq i < N\ \} \\
\cup\ &\{\ [t,2,i\ ] \rightarrow \texttt{cur}[i] \mid 0 \leq t < T \wedge 1 \leq i < N-1\ \}
\end{aligned}$$

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 on the right 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

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

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 [16].

The storage mapping used for general uniprocessor time skewing is

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

The `tide` arrays are used to hold the values that are live across blocks in the non-time-step loops. The storage mappings required to provide separate arrays for the inter-subtile boundaries for multiprocessor time skewing can be

produced in a straightforward (if somewhat tedious) application of the subtile constraints to the storage mapping above.

8 Related Work

Most current techniques for improving locality [2, 20, 19, 10, 3, 4] 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, as the inner loops often have finite compute balance.

Pugh and Rosser [14] 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 [11], 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 [17] 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 tile size. However, Song and Li only discuss optimizations for uniprocessors.

Högstedt, Carter, and Ferrante [3, 4] have developed an algorithm for tiling to minimize the idle time of the processor. Their algorithm is only applicable to perfectly nested loops, and considers only atomic tiles. Figure 7 shows the tiling their algorithm would produce for Figure 1. The gray bars represent wavefronts (these are the same as ours). Each processor receives data sent earlier by the processor to its north, executes a tile (according to the wavefront shown), then sends the values across the southeastern border.

If the leading wavefronts fit in L1 cache, this tiling will produce no main memory traffic beyond what is needed for communication. Let η be the number of wavefronts in a tile. To eliminate idle time in our example code using this tiling,

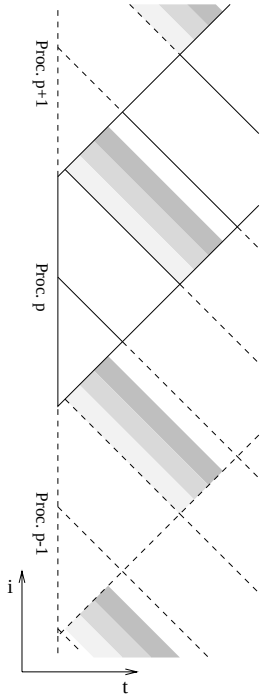

Figure 7. Tiling Figure 1 as in [4]

we must choose η such that $\frac{O}{C}\eta^2 \geq L + \frac{D}{B_N}\eta$ (that is, ensure that execution time for a tile is at least as great as communication time). If three wavefronts are to be kept in cache, it requires an L1 cache of size $3D\eta$. On our hypothetical machine, the minimum η is 306, requiring 7344 bytes of L1 cache (the minimum L1 cache for our algorithm is 1440 bytes, though this requires more iterations per processor).

For some architectures, the tiling of Figure 7 has a lower minimum cache requirement. In general, $\eta \geq \frac{DC + \sqrt[+]{D^2C^2 + 4OB_N^2CL}}{2OB_N}$. Thus, the cache requirement for Figure 7 ($3D\eta$) will be lower than than given in Section 3 ($\frac{6D^2C}{OB}$), iff $B(DC + \sqrt[+]{D^2C^2 + 4OB_N^2CL}) \leq 4DCB_N$.

Attempting to compare the amount of parallelism allowed by the two approaches yields an interesting insight. To maximize parallelism, we minimize the "height" of the tiles (σ) in Section 3. We can shrink σ and increase τ until $\sigma = \tau$, at which point the white "central region" of Figure 3 disappears. If we do this, and then "transfer the data in many small blocks" (as we suggest for small L), we get a tiling that is like Figure 7, but periodically includes tiles that all end at the same time step (this is useful for checkpointing and backing out of while loops that have executed too many iterations). Thus, the two tilings are almost identical in this extreme case.

A preliminary version of the multiprocessor time skewing algorithm was presented as a poster at LCPC '99.

9 Conclusions

Uniprocessor time skewing can be used to eliminate idle time due to high machine balance, given a sufficient number of time steps and an L1 cache of a size that is a function of the machine 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 a sufficiently large problem and sufficient 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 allow idle time waiting for main memory, (b) use tiles that fit in L2, possibly creating idle time waiting for L2, or (c) use multi-level tiling to prevent idle time altogether, given sufficient bandwidth to L2 and a smaller amount of L1 cache.

Acknowledgements

We would like to thank Larry Carter, Karin Högstedt, Michelle Mills Strout, the other participants and reviewers at LCPC '99, and the reviewers for IPDPS '00, for their thoughts about this work.

This work is supported by NSF grant CCR-9808694.

References

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

[2] 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.

[3] K. Högstedt, L. Carter, and J. Ferrante. Determining the idle time of a tiling. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 160–173, Jan. 1997.

[4] K. Högstedt, L. Carter, and J. Ferrante. Selecting tile shape for minimal execution time. In *roceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, pages 201–211, June 1999.

[5] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, Dept. of Computer Science, University of Maryland, College Park, Mar. 1995. The Omega library is available from http://www.cs.umd.edu/projects/omega.

[6] W. Kelly and W. Pugh. Determining schedules based on performance estimation. *Parallel Processing Letters*, 4(3):205–219, Sept. 1994.

[7] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *The 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, McLean, Virginia, Feb. 1995.

[8] J. McCalpin and D. Wonnacott. Time skewing: A value-based approach to optimizing for memory locality. Technical Report DCS-TR-379, Dept. of Computer Science, Rutgers U., Feb. 1999.

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

[10] K. 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.

[11] T. C. Mowry, M. S. Lam, and A. 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, Oct. 1992.

[12] W. Pugh. Counting solutions to presburger formulas: How and why. In *SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[13] W. Pugh and D. 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/.

[14] E. J. Rosser. *Fine-Grained Analysis of Array Computations*. PhD thesis, Dept. of Computer Science, The University of Maryland, Sept. 1998.

[15] T. Shen, J. Spacco, and D. Wonnacott. High MFLOP rates for out of core stencil calculations using time skewing. In *SC '97 poster session*, Nov. 1997. Available as http://www.haverford.edu/cmsc/davew/cache-opt/SC97poster.ps.

[16] T. Shen and D. Wonnacott. Code generation for memory mappings. In *The 1998 Mid-Atlantic Student Workshop on Programming Languages and Systems (MASPLAS '98)*, Apr. 1998. An updated version is available as http://www.haverford.edu/cmsc/davew/cache-opt/mmap.ps.

[17] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 215–228, May 1999.

[18] M. Spivak. *Calculus*. Publish or Perish, Inc., Berkeley, CA, 1980.

[19] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of Computer Science, Stanford U., Aug. 1992.

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

[21] D. Wonnacott. Achieving scalable locality with time skewing. Technical Report DCS-TR-378, Dept. of Computer Science, Rutgers U., Feb. 1999.

[22] D. Wonnacott. Achieving scalable locality with time skewing. 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., 2000.