A General Algorithm for Time Skewing

David Wonnacott
Haverford College
davew@cs.haverford.edu

July 13, 2001

Abstract

Microprocessor speed has been growing exponentially faster than memory speed in the recent past. In this paper, we consider the long term implications of a continuation of this trend. We define a program property known as scalable locality, which measures our ability to apply ever faster uniprocessors to increasingly large problems (just as scalable parallelism measures our ability to apply more numerous processors to larger problems). We then show that exploiting scalable locality in scientific programs often requires advanced compile-time reordering of loop iterations. We provide an algorithm that derives an execution order, storage mapping, and cache requirement for any desired degree of locality, for certain programs that can be made to exhibit scalable locality. Our approach is unusual in that it derives the program transformation and cache requirement from the dataflow of the calculation (a fundamental characteristic of the algorithm), instead of searching a space of possible transformations of the execution order and array layout used by the programmer (which are artifacts of the expression of the algorithm). We include empirical results showing the effectiveness of our transformation on two small kernels and a non-trivial benchmark program. Our transformation can produce speedups for data sets residing in L2 cache, main memory, or virtual memory.

This report unifies the major results of DCS-TR-379 and DCS-TR-378, presents them in terms of a general algorithm rather than a collection of heuristics, and gives a clearer presentation of the empirical data.

Keywords: Compute balance, loop skewing, machine balance, memory locality, scalable locality, storage transformation

1 Introduction

Microprocessor speed has been growing exponentially faster than main memory speed in the recent past [McC95]. The possibility that this trend may continue raises the question of whether or not there is an upper limit on useful processor speed: at some point, will memory bandwidth limits make additional gains in processor speed irrelevant?

In this paper, we define *scalable locality*, which corresponds to our ability to apply ever faster uniprocessors to increasingly large problems, without requiring a cache that grows with the problem size. This is analogous to the use of the term scalable parallelism to describe code for which we can apply greater numbers of processors to larger problems.

We investigate the problem of exposing scalable locality in scientific programs (i.e. programs that use calculations in nested loops to manipulate large arrays of data). For some calculations, loop tiling [Wol89] is sufficient to expose scalable locality. However, for other calculations, additional transformations such as loop skewing are required as well. This result is somewhat surprising, as Wolf and Lam found that loop skewing did not play a significant role in improving locality in practice [WL91, Wol92]. More recent studies, such as those by McKinley et al. [MCT96] and Roth et al. [RMCKB97], have ignored loop skewing on the grounds that Wolf and Lam found it was not useful. However, some calculations benefit from loop skewing only after other advanced transformations not studied by Wolf and Lam have been applied. This is reminiscent of the search for scalable parallelism, where complex analysis and transformations are required before parallelization can be applied [EHLP91, EHP98].

```
for (int i = 0; i<N; i++)
  for (int j = 0; j<N; j++)
    old[i][j] = cur[i][j]
for (int i = 1; i<N-1; i++)
  for (int j = 1; j<N-1; j++)
    cur[i][j] = 1.0/8 *
                        (old[i-1][j] +
       old[i][j-1] + 4*old[i  ][j] + old[i][j+1] +
                         old[i+1][j]);
```

Figure 1: Five Point Stencil

We have developed techniques to determine whether or not a given calculation can be made to display scalable locality, and to derive, for certain codes, transformations of the execution order and storage mapping to produce any arbitrarily high degree of locality. Since the transformations are derived from the underlying dataflow, they can automatically produce high spatial as well as temporal locality, and avoid cache interference. Our algorithm also produces information about the cache size required for a given degree of locality.

The rest of this paper is organized as follows: We give our definition of scalable locality and introduce the notation we use for program transformations in Section 2. We also review the work of Wolf and Lam in this section, relating their techniques to our definitions and notation.

We introduce our techniques by demonstrating their application to two simple stencil calculations, and then present our general algorithm. In Section 3, we define a class of programs known as time-step calculations, which includes programs outside the domain of other locality optimization techniques. We then give our algorithm for transforming iteration spaces of these calculations to permit scalable locality; We use the term *time skewing* for this transformation, as it generally involves skewing with respect to the time-step loop. In Section 4, we discuss transformations of the mapping of values to memory locations. For some programs, these transformations are required to produce scalable locality; for others, these transformations may optionally be used to minimize memory traffic. We then present our general algorithm in Section 5, review its application to our stencil codes, and discuss its use for a larger example from a benchmark.

We give empirical results in Section 6. Our transformation can produce speedups for data sets residing in L2 cache, main memory, or virtual memory. For our three-point stencil, the transformed code can process data sets residing in L2 cache or main memory at 90% of the peak speed achieved by the original algorithm (for a data set residing in L1 cache), or data sets residing in virtual memory at about 70% of this peak speed (the original code runs at about 30% of peak for main memory, or 0.1% of peak for virtual memory). We discuss other approaches to improving locality in Section 7, and give concluding remarks in Section 8.

2 Machine Balance, Compute Balance, and Scalable Locality

In the discussion that follows, we consider the relative speeds of a processor and memory system in light of the demands made of each when running a given piece of code. We use the term *machine balance* [CCK88, McC95] to refer to the ratio of the maximum sustainable rate at which a processor can perform floating-point arithmetic (typically for data in registers) to the maximum sustainable transfer rate for unit-stride accesses for a memory system (in units of floating point values per second).

We wish to measure the fundamental resource requirements of a piece of code in a way that is independent of the compile-time optimizations applied to it and the architecture used to execute the code. Thus, we define the *compute balance* of a piece of code as the ratio of the number of required floating point operations to the number of floating point values that are live at the entry or exit of the code. We refer to values whose live ranges are entirely contained within the code under consideration as temporaries.

In general, compute balance can vary with number of statements being considered, or the number of loops surrounding the statements. In the absence of common subexpressions or loop invariant computations, the compute balance of a piece of code will be no less than the average compute balance of subsections of the

```
// initialize C to zero
for (int i = 0; i<N; i++)
  for (int j = 0; j<N; j++)
    for (int k = 0; k<N; k++)
      C[i][k] += A[i][j] * B[j][k];
```

Figure 2: Matrix Multiplication

code: values that flow only between the subsections will no longer be considered, but all operations in both subsections must still be performed. For example, a single execution of the body of the loop in Figure 1 has a compute balance of 1 (it reads 5 values from `old`, produces 1 value in `cur`, and performs 2 multiplications and 4 additions); two executions of this statement for consecutive values of `j` have combined compute balance of $\frac{12}{10}$; a complete execution of the `j` loop has a balance of approximately 1.5 (it reads about[1] $3N$ values, writes about N values, and performs about $6N$ operations); and the entire `i/j` loop nest has a balance of approximately 3.

By comparing compute balance to main memory machine balance, we can determine whether we might obtain full processor utilization when non-temporaries are stored in main memory. When the machine balance is greater than the compute balance, the total time required by the accesses to main memory exceeds the total processor time required, and the code is fundamentally bandwidth-bound (regardless of the number and fate of temporary values, issues such as cache interference, or latency hiding optimizations such as prefetching). In this case, the only way to improve processor utilization to ensure that some of the values used or produced by the calculation are in cache: that is, by optimizing for cache locality on a scale larger than the code currently being considered.

When the compute balance is greater than the machine balance, the required processor time is greater than the main memory access time for non-temporaries. In this case, we optimize locality by focusing on the placement of temporaries produced within the piece of code. In practice, this does not guarantee efficient processor use—other factors, such as memory latency, cache interference, or inefficient use of bandwidth due to lack of spatial locality, may interfere. In such cases, it may be possible to counter these factors with compile-time optimizations: Copying can be used to reduce cache interference [LRW91, TGJ93], and prefetching can reduce stalls due to latency [MLG92]. We ignore these effects in our analysis and transformation. In principle, we could apply techniques to address these effects after performing our optimization, but we have not found it necessary to do so, as our full algorithm dramatically reduces their significance.

Locality optimization should be performed on a scope that is large enough for the compute balance of the code to be at least as great as the main memory machine balance, since full processor utilization requires this condition. In some cases, the entire program may have a finite compute balance, and no amount of optimization can produce efficient use of CPU that is far faster than memory. In other cases, the compute balance of a loop nest or program may be a function of the program parameters. For example, the complete loop nest for matrix multiplication (Figure 2) has compute balance of $\frac{2N}{3}$: it reads $2N^2$ values, produces N^2 values, and performs N^3 multiplications and N^3 additions. We say that such calculations exhibit *scalable compute balance*: the compute balance increases with the number of operations performed.

2.1 Scalable Locality

As we have noted, achieving high compute balance does not guarantee effective cache use. (In the terms used by Wolf and Lam, "reuse does not guarantee locality" [WL91].) We define the *average locality* of a calculation on a particular machine as the number of floating-point operations per floating-point value transferred to or from main memory. If only the live values are written to (and read from) main memory, then the average locality will equal the compute balance. Average locality can exceed compute balance if some live values are already in cache at the start of the calculation, but this effect should be negligible at large scopes, and we will not count on it in our analysis. If main memory is used for temporaries, average locality may fall below the compute balance (possibly by more than a constant factor).

[1] Here, we use *about* as defined by Sedgewick [Sed98, Chapter 2.4]: $F(N)$ is *about* $f(N)$ if $F(N) = f(N) + g(N)$, where $g(N)$ is asymptotically small compared to $f(N)$.

```
// initialize C to zero
for (int jj = 0; jj<N; jj+=s)
  for (int kk = 0; kk<N; kk+=s)
    for (int i = 0; i<N; i++)
      for (int j = jj; j<min(N, jj+s); j++)
        for (int k = kk; k<min(N, kk+s); k++)
          C[i][k] += A[i][j] * B[j][k];
```

Figure 3: Tiled Matrix Multiplication (from [WL91])

Temporaries may end up in main memory because of limits on cache size or associativity, or simply because old values are generally written out of cache when a line is reallocated, even if all the old values are dead. We can reduce the use of main memory for temporaries by using a single block of addresses for all temporaries. If this block is small enough to reside in cache, there is (in the absence of interference from accesses to non-temporaries) no need to write any temporary to main memory. If the target architecture does not allow us to prevent interference due to non-temporaries, we simply ignore this effect, as the vast majority of our memory references will be to the temporary block. (In principle, we could compensate for this interference (and for bandwidth that is wasted on partially used cache lines) by adjusting our estimate of the amount of memory traffic produced by each non-temporary (or non-unit-stride) access.)

Note that the temporary block must be at least as large as the number of simultaneously live temporaries. Thus, to keep locality from falling below the compute balance, we must control the number of simultaneously live temporaries, and keep our block of memory for temporaries close to this size, so that it will fit in cache. We say that we can produce *scalable average locality*, or simply *scalable locality*, if we can increase the average locality of the calculation with the number of operations performed, without requiring a cache that grows linearly with the problem size. (If our definition did not place any restriction on cache size, we could claim scalable locality for any number of programs by simply requiring a cache big enough to hold the entire data set and all temporaries. This is neither interesting nor useful.) We say that scalable locality has been produced *for a fixed cache size* if the cache requirement does not grow at all with the problem size (though the cache requirement may depend on the degree of locality produced).

We approach the search for scalable locality in three steps: First, we examine larger and larger regions of the program, until we either identify one with scalable compute balance or fail (having exhausted the entire program or run into excessive function call inlining). If we find a section of code with scalable compute balance, we attempt divide this code into an ordered sequence of tiles such that (a) executing the calculation tile-by-tile produces the same result, (b) the balance of each tile can be scaled up (possibly by increasing the tile size), and (c) the calculations in each tile are executed in an order such that the number of temporary values that are simultaneously live grows less than linearly with the problem size. We say that such a tiling produces an *ordering that permits scalable locality.* If we can produce such an ordering, we then attempt to find a way to map the temporaries to addresses in such a way that all temporaries are stored in a set of arrays (which we call "cache" arrays) whose total size grows less than linearly with the problem (property c makes this possible). We call the mapping from the statement and iteration that produces a value to the address used to store it the *storage mapping.* Our goal, when transforming the storage mapping, is to ensure that the number of temporaries written to memory does not grow with the problem size. In many cases, we can produce scalable locality even if we omit this last step and use the original storage mapping, though this does force more temporaries to be written to memory.

For a perfectly nested set of loops, such as matrix multiplication, traditional loop tiling [Wol89] may produce scalable locality, as shown in Figure 3. The `jj` and `kk` loops enumerate $(\frac{N}{s})^2$ tiles, each with balance $\frac{2Ns^2}{2Ns+s^2}$, so we can scale compute balance up by increasing s. Within each tile, the total number of temporaries that are live simultaneously does not exceed s^2+2s (s^2 elements of `B` and s of `A` and `C`), so we have achieved scalable locality for fixed cache size $O(s^2)$ (assuming other optimizations are used to prevent cache interference).

Note that tiling may be useful even when it does not produce scalable locality. For example, the `i/j` loop nest of Figure 1 does not have scalable balance, and thus cannot have scalable locality. Tiling can still

```
for (int t = 0; t<T; t++)
  for (int i = 0; i<N; i++)
    A[i+1] = 1.0/3 * (A[i] + A[i+1] + A[i+2]);
```

Figure 4: Three Point In-Place Stencil (from [WL91])

```
for (int ii = 0; ii<T+N; ii+=s)
  for (int t = 0; t<T; t++)
    for (int i = max(t,ii); i<min(t+N,ii+s); i++)
      A[i+1-t] = 1.0/3 * (A[i-t] + A[i+1-t] + A[i+2-t]);
```

Figure 5: Tiled Three Point In-Place Stencil (from [WL91])

improve its locality by a fixed amount. In this paper, we only consider techniques for producing scalable locality.

Before discussing our techniques, we review one of Wolf and Lam's examples [WL91], for which skewing and tiling must be combined to produce scalable locality. This review also serves to introduce our style of diagrams. We then discuss the transformation system used by Kelly and Pugh [KP94], which we will use (and extend) for our system.

2.2 Loop Skewing and Tiling

Wolf and Lam [WL91] show that other transformations may be required before tiling a loop nest. For example, to tile Figure 4, we must skew [Wol89] the `i` loop first. (The resulting code is shown in Figure 5.)

Figure 6 shows how the tiles of the code in Figure 5 group the loop iterations, for $T = 6$, $N = 7$, and $s = 2$ (as in [WL91, Figure 2]). Each circle represents the execution of one iteration, and each arrow a value-based flow dependence (a dataflow) [PW93, PW98]. Tiles are indicated by dashed lines: all iterations for which $ii = 0$ are in the triangle at the lower left, those for which $ii = 1$ are in the trapezoid next to it, etc. Dependences constrain the order of execution of the tiles; we must execute them from lower left to upper right, as is done in Figure 5. For this calculation, the memory-based dependences (often simply called "dependences" [Wol89]) are a subset of the closure of the value-based flow dependences, so preserving dataflow is sufficient to maintain the program semantics. In other cases, we must preserve all dependences or change the way values are stored in memory, to maintain program semantics.

Figure 6 also serves to illustrate the compute balance of each tile graphically: The number of operations in a tile is indicated by the number of circles (each of which represents one iteration, or three operations); the number of live values entering and exiting the tile is indicated by the number of circles from which an arrow crosses the tile borders (this is essentially half the number of arrows crossing the borders – except at the edge of the calculation, the iterations at the tile border send their data to two iterations in the next tile). Thus, the number of live values is proportional to the *length* of the tile (from upper left to lower right), while the number of operations is proportional to the *area* of the tile, making the balance proportional to the thickness of the tiles, s. We will note this contrast between the n dimensional tile volume and the $n-1$ dimensional tile border in many other examples; we have found that identifying it visually provides significant insight into our techniques.

The shading in Figure 6 shows a possible state of the calculation, and thus indicates the ordering of iterations within a tile: recently completed iterations are shaded gray, with the most recent iteration the darkest. Note that the iterations that precede the gray "wavefront" do not produce any values that will be needed in the part of the tile after the gray region. Thus, the number of live temporaries grows with the tile thickness (s), but not the problem size (N or T), and thus this tiling yields an ordering that permits scalable locality for fixed cache size (since a cache of size $O(s)$ is needed). More precisely, there are never more than s live temporaries: The values created by the iterations on the right or upper right edges of tiles are live past the end of the tile; all other values live at most s iterations, since the value defined in iteration $[t, i]$ is used only in iterations $[t, i+1]$ (one iteration later) $[t+1, i-1]$, and $[t+1, i]$ ($s-1$ and s iterations

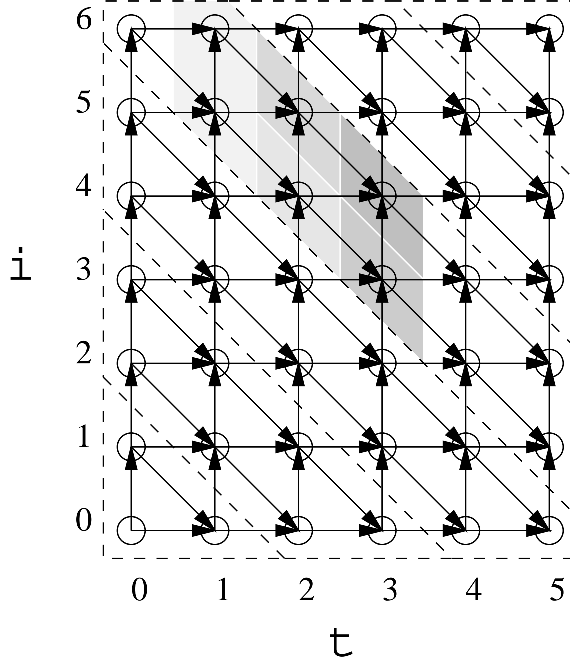

Figure 6: Tiled Iteration Space for Three Point In-Place Stencil

later, respectively).

For this code, the original memory mapping can produce scalable locality for this ordering of the calculation. From the assignment statement, we can see that the value produced in iteration $[t, i]$ will be stored in `A[i+1]`. Thus, each vertical column of iterations in a tile overwrites all of the temporaries written in the previous column; only the one non-temporary is not overwritten. In the absence of significant cache interference, the temporaries will remain in cache until they are overwritten, and only the live values will be sent to main memory.

The tiling shown by Wolf and Lam is not the only tiling that permits scalable locality. Figure 7 shows an alternate ordering for which each tile ends at a constant value of the outer loop index, a property that we will make use of when we discuss loops with early exits in Section 5.

2.3 Describing Transformations

The transformations used here can be described concisely in the framework of Kelly and Pugh [KP94]. In this framework, each execution of each statement is identified with a unique tuple of integers. For example, we can refer to the iteration of Figure 1 in which $t = 3$ and $i = 7$ as $[3, 7]$. We can describe the set of all iterations with a set of constraints derived from the loop bounds and conditional expressions [PW93, PW98], for this example,

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

The integers in these tuples may correspond to the loop index values of surrounding loops (as above), or they may indicate which of several statements is being performed. For example, if there were two statements in this loop, we could refer to the executions of individual statements as $[t, i, 1]$ and $[t, i, 2]$; statements in two consecutive nests would be $[1, t, i]$ and $[2, t, i]$.

The lexicographical ordering of these tuples defines the execution ordering of the statements and iterations, so we can describe a reordering transformation as a remapping of the tuples assigned to the iterations. The transformation used by Wolf and Lam to produce Figure 5 is

$$\{ \; [\; t, i \;] \rightarrow [\; (t+i) \; div \; s, t, (t+i) \; mod \; s \;] \; \}$$

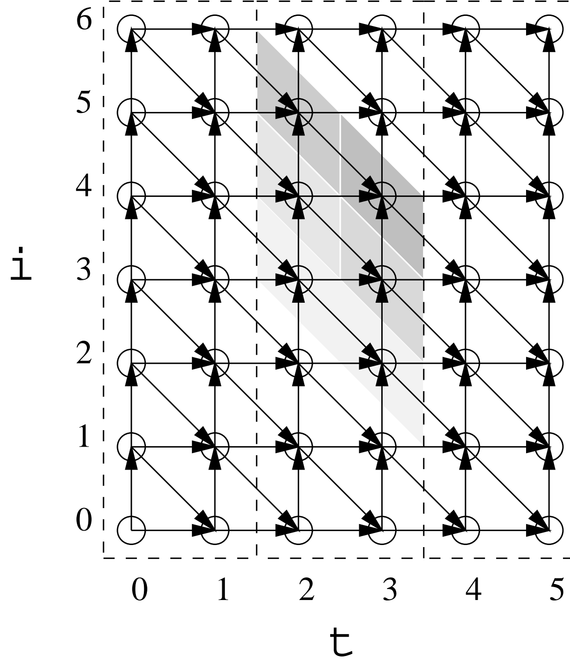

Figure 7: Alternative Tiling for Three Point In-Place Stencil

and the transformation for Figure 7 is

$$\{ [t, i] \rightarrow [t \; div \; s, i + t, t \; mod \; s] \}.$$

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). In the first transformation, the tiles are skewed; in the second, the tiles are not skewed, but the execution of iterations within each tile is.

We can use the constraint manipulation techniques discussed by Pugh and Wonnacott [PW93, Won95] to check that the transformation does not violate any data dependences, and to apply the transformation to the set describing the iteration space. This transformed set can then be used as input to the code generation algorithms of Kelly et al. [KPR95].

The constraint manipulation and code generation algorithms require that all constraints be affine, which requires that s must be a known constant in the transformation above. More details about the use of this system can be found in the Omega Library documentation [KMP+95]; algorithmic details can be found in the papers cited above.

3 Time Step Calculations and Iteration Space Time Skewing

We say that a calculation is a *time-step calculation* if it consists entirely of assignment statements surrounded by structured `if`'s and loops (possibly `while` loops or loops with `break` statements), and all loop-carried value based flow dependences come from the previous iteration of the outer loop (which we call the time loop). For example, if we add a time step loop around the stencil in Figure 1, we get the time step calculation shown in Figure 8. The simpler three point stencil calculation in Figure 9 also fits our definition of a time-step calculation, though the "in-place" version in Figure 4 does not: the value computed in iteration $[t, i]$ is used in iteration $[t, i + 1]$ (as `A[i]`).

If only the values from the last time step are live after the end of the time loop, and all values that are live on entry to the calculation are read in the first time step, the balance of a time-step calculation is proportional to the number of time steps. Thus, we may be able to achieve scalable locality for such loop nests by producing tiles that combine several time steps. Unfortunately, current approaches to loop tiling

```
for (int t = 0; t<T; t++)
  {
  for (int i = 0; i<N; i++)
    for (int j = 0; j<N; 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 8: Iterative Five Point Stencil

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

Figure 9: Three Point Stencil

cannot tile these time-step calculations: the techniques of Wolf and Lam cannot be applied to a loop that contains several nests (the assignments to `old` are implicit loops), and more recent work [MCT96, RMCKB97] has not used loop skewing (on the grounds that Wolf and Lam did not find it useful in practice).

In this section we show to produce an ordering that permits scalable locality for any time-step "stencil" calculation (a calculation in which each array element is replaced by a function of the previous values of its nearest neighbors). Section 5 discusses the generalization of our algorithm to a wider class of calculations. We call the transformation that produces this ordering, in which tiles end at a uniform value of the time step loop and all other loops have been skewed with respect to the time step loop, *time skewing*.

We start by performing value-based dependence analysis [PW93, PW98]. The result of this analysis is a graph of the flow of values through the iteration space. We then perform array expansion, followed by copy propagation for any array copy [Won00a], converting the stencil into a single set of perfectly nested loops in "functional style" (as shown in Figure 10).

We then tile the iterations of the new code to permit scalable locality. For the three point stencil, we use the transformation

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

producing the code shown in Figure 11 and the iteration ordering shown in Figure 12. This ordering of iterations permits scalable locality, much as it did in Figure 7 for the "in-place" code. As before, the number of operations grows with the volume of the tile (Ns iterations of 4 operations each), and the number of live

```
// define cur[1:T-1][0] to be equal to cur[0][0],
// and  cur[1:T-1][N-1] to be cur[0][N-1].
for (int t = 0; t<T; t++)
  for (int i = 1; i<=N-2; i++)
    cur[t+1][i] = 0.25 * (cur[t][i-1] + cur[t][i]+cur[t][i] + cur[t][i+1]);
```

Figure 10: Three-Point Stencil in Functional Style

```
#define s2(t,i) cur[(t)+1][i]=0.25*(cur[t][(i)-1]+cur[t][i]+cur[t][i]+cur[t][(i)+1])

for(t2 = 0; t2 <= intDiv(T-1,8); t2++)
  for(t3 = 8*t2+1; t3 <= min(N+8*t2+5,N+T-3); t3++)
    for(t4 = max(-N+t3-8*t2+2,0); t4 <= min(T-8*t2-1,t3-8*t2-1,7); t4++)
      s2(t4+8*t2,t3-t4+-8*t2);
```

Figure 11: Three-Point Stencil after Time Skewing (for s=8)

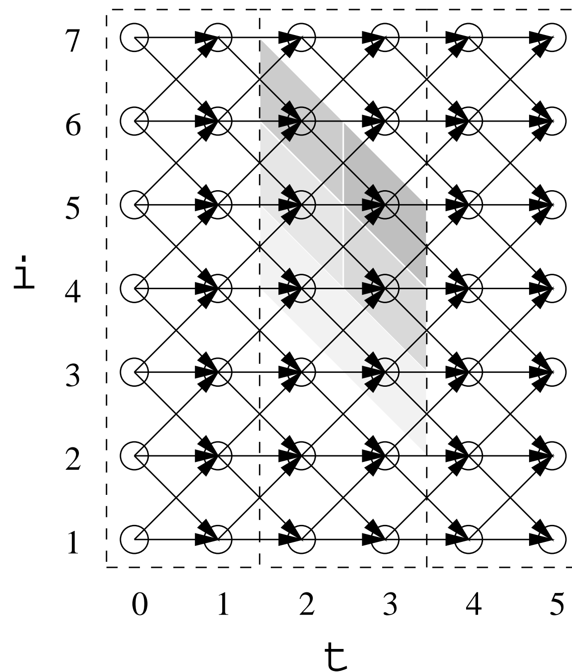

Figure 12: Time-Skewed Iteration Space for Three Point Stencil

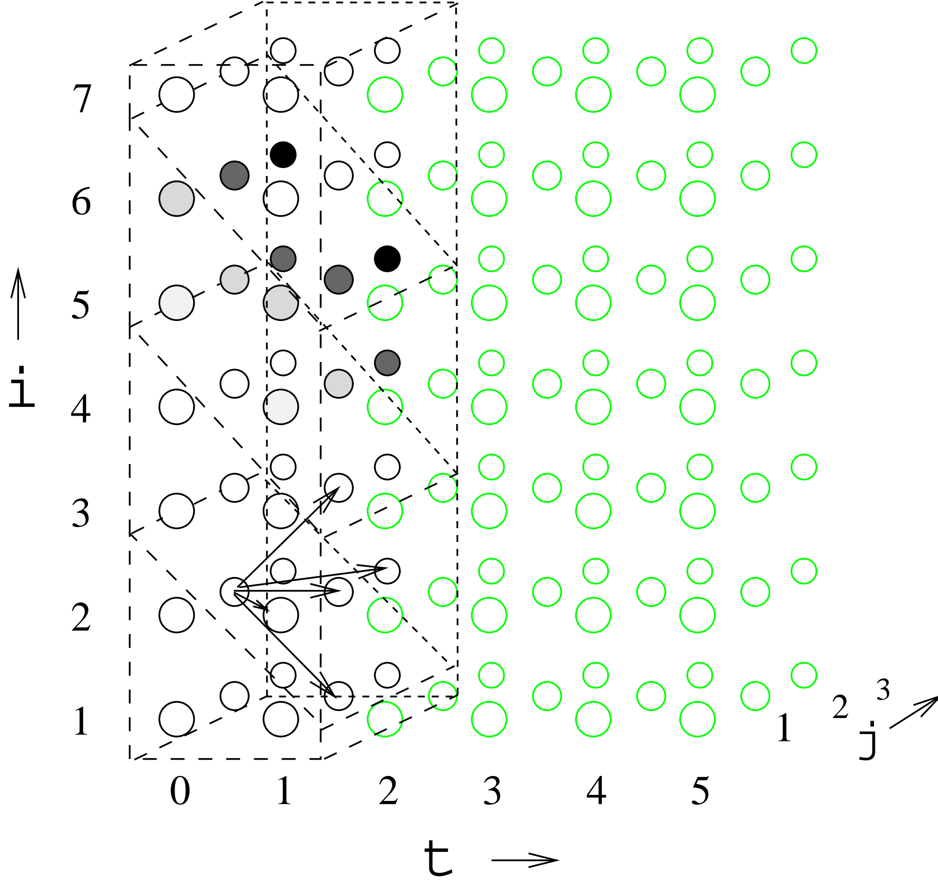

Figure 13: Time Skewed Iteration Space for Five Point Stencil

values with the size of the tile's border (N live values are read and N are produced). Thus, the balance of each tile is $2s$. As only $2s + 1$ values are live simultaneously during the execution of a tile, the ordering permits scalable locality for fixed cache size.

We can determine the minimum value of s from the compute balance of the tiles ($2s$) and the balance of the target machine. For example, we would set $s \geq 15$ for machine with balance 30, or $s \geq 250$ for machine with balance 500 (such as the virtual memory system used in Section 6). In practice, we may choose a larger s if it will fit in cache, to minimize loop overhead. As we will see in 4.3, we require enough cache for $3s$ values for this example, or about $6K$ bytes for $s = 250$.

The transformation for the five point stencil in Figure 8 is slightly more complicated, though the basic approach remains the same: We first perform dataflow analysis to find the compute balance of each loop. The resulting iteration space (after copy propagation) and dataflow are shown Figure 13. Several simplifications have been made to this figure: dataflow arrows are shown from only one example iteration (uniform arcs exist throughout the iteration space); only the first three iterations of the `j` loop are shown; tiles are shown only in the first time block (for $s = 2$) – the tiling pattern is repeated for time steps 2 and 3 and for time steps 4 and 5 (these iterations have gray borders); and only four levels of gray are used to indicate the ordering of the twelve iterations within a tile.

The tiling results from our search for tiles that permit scalable locality. We must block the time loop, since no inner loop has scalable balance. Blocking only this loop would produce tiles whose balance grows with the tile size, but this would not let us set a fixed limit on the number of simultaneously live temporaries (any wavefront intersects such a tile in an area that grows at least linearly with N). We therefore block the time and `i` loops, as shown in the figure. (We use s_t and s_i to refer to the block sizes for the `t` and `i` loops when they differ.) Note that dependences require a skewed tile border in the `i` dimension.

The balance of these tiles grows with $\frac{3s_is_t}{s_i+2s_t}$: each tile performs Ns_ts_i iterations of 6 operations, and Ns_i has live values at the left and at the right and $2Ns_t$ live values at the top and at the bottom (the dependences indicate that values one layer inside this border are live across it). To derive s_i and s_t for a particular balance, we note that the balance approaches $min(3s_t, \frac{3}{2}s_i)$ as $\frac{s_t}{s_i}$ approaches 0 or infinity. We can produce high balance with lower cache requirements (see below) by setting $s_t = 2s_i$, in which case the balance is $3s_i$ (this same result can be derived from the derivative of the balance equation). For example, on a machine with balance 30, we would set $s_t = 20$ and $s_i = 10$; with machine balance 500, $s_t = 333$ and $s_i = 167$

The wavefront of execution within a tile is skewed with respect to both `i` and `j`, to preserve dependences (iterations within a wavefront must be executed in order of increasing t). This wavefront intersects each tile over an area of s_ts_i iterations; given the dependences, we have just over 2 wavefronts of live temporary values at any given time. Keeping 3 wavefronts in cache requires a cache of size $24s_ts_i$ (with 8-byte real numbers), Thus, about 4.8K of cache is needed for a balance of 30, but almost 1.3M of cache is needed for a balance of 500.

This tiling and wavefront are produced with the transformation

$$\{\ [\ t, i, j\ ] \rightarrow [\ t\ div\ s_t, (t+i)\ div\ s_i, t+i+j, t\ mod\ s_t, (t+i)\ mod\ s_i\ ]\ \}.$$

The *div* terms define the blocks in the time (t) and skewed `i` ($t+i$) dimensions, and the $t+i+j$ term defines the wavefront within the tile.

For stencils of larger dimension that involve only nearest neighbors, we generalize the above transformation as

$$\begin{aligned}\{\ [t, i_1, i_2 ... i_n] \rightarrow [\ & t\ div\ s_t, \\ & (i_1+t)\ div\ s_1, (i_2+t)\ div\ s_2, ...(i_{n-1}+t)\ div\ s_{n-1}, \\ & t + \Sigma_{l=1}^{n} i_l, \\ & (i_1+t)\ mod\ s_1, (i_2+t)\ mod\ s_2, ...(i_{n-1}+t)\ mod\ s_{n-1}, \\ & t\ mod\ s_t\ ]\ \}\end{aligned}$$

Once again, we must block the time step to create scalable balance, and block all but one spatial dimension to ensure that the intersection between the wavefront and tile does not grow with the problem size. Dependences force us to skew the wavefront in each spatial dimension. For nearest-neighbor dependences, we will generally keep $3s_t\Pi_{i=1}n - 1s_i$ values in cache, and two "layers" of iterations will be live between the blocks in the spatial dimensions (so $\forall_i, s_t = 2s_i$). Thus, the amout of cache required to produce balance b for a d-dimensional nearest-neighbor stencil generally grows with b^d.

For stencils with different dependence patterns, we will need different skew factors, and use varying aspect ratios for the tiles. This can be addressed by using prior work on skewing for perfect loop nests [WL91], or the techniques we introduce in Section 5 for other cases.

This transformation produces an iteration space that permits scalable locality. However, if we write the temporary values into the expanded array, the code will exhibit extremely poor cache performance.

4 Storage Mappings and Time Skewed Storage Mappings

In this section, we consider the task of mapping temporary values to storage locations in a way that achieves scalable locality, given an iteration space that permits scalable locality. We give two simple mappings that produce locality that is scalable but not optimal (in the sense that many temporaries must be written to main memory), and a more complex mapping for which the fraction of the temporaries written to main memory approaches zero as the problem size grows.

We refer to these mappings of values to memory locations as *storage mappings*. They can be described in terms similar to those we use for iteration space transformations: To describe the mapping from the values produced by the iterations of a given statement to a given array, we use a relation from the iteration space of the statement to the array indices. We can use a separate storage mapping for each statement, and in some cases may use a union of relations to describe a mapping of the values produced by one statement to a collection of different arrays. We can describe these mappings and partial mappings in terms of affine constraints, and thus use the extensions to the Omega Library described by Shen and Wonnacott [SW98] to generate the transformed programs.

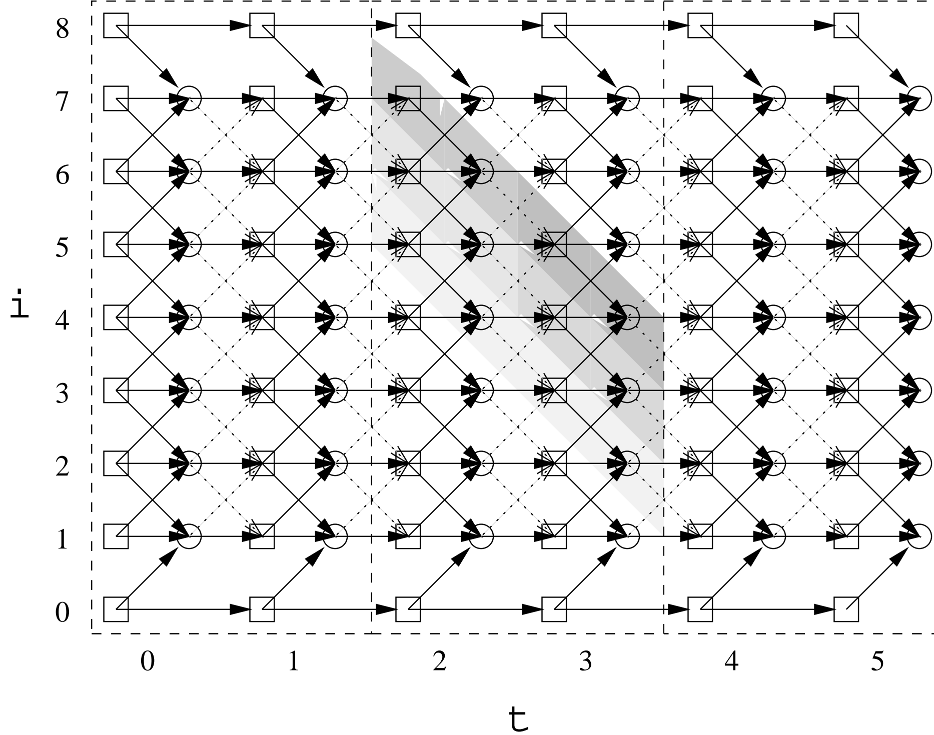

Figure 14: Tiled Iteration Space for Three Point Stencil

4.1 Original Storage Mapping

It is possible to use the original storage mapping for the programs in Figures 8 and 9. Doing so requires a slight modification to the iteration space transformation we perform, as the time-step loop once again contains two nests (for the array copy and for the calculation). The iteration space for the three point stencil, and the transformation we perform, are shown in Figure 14. The column of squares represents the array copy, and a column of circles represents the calculations. In this calculation, there are anti-dependences (shown as dotted lines) that are not redundant with respect to the dataflow, and thus must be obeyed if we are not transforming the storage mapping.

We perform this transformation by first "renumbering" the iterations so that each column in Figure 14 has a unique value of t, and then considering memory-based dependences as well as dataflow dependences when performing time skewing. That is, we first remap iteration $[t, w, i]$ to $[2t + w, i]$, where w is a constant distinguishing between the two inner loops: 0 for the statement in the array copy, or 1 for the computation statement; and then apply the transformation we used for Figure 12. The combined transformation is

$$\{ [\ t, w, i\ ] \to [\ (2t + w)\ div\ 2s, i + 2t + w, (2t + w)\ mod\ 2s\ ] \}.$$

Note that this transformation does not use the same "angle" of skewing as time skewing: For the three point stencil, calculation $[3, 4]$ is preceded by calculation $[2, 5]$ in Figure 12, and by calculation $[2, 6]$ (and assignment $[2, 5]$) here.

For a fixed s, this code can be generated by the Omega Library, as shown in Figure 15. Since we have rearranged the iteration space, but not the storage mapping, it is not surprising that we can produce this transformation with a combination of traditional transformations, specifically: loop alignment, imperfectly nested loop interchange, loop skewing, and tiling.

```
#define s2(i) old[i]=cur[i]
#define s3(i) cur[i]=0.25*(old[(i)-1]+old[i]+old[i]+old[(i)+1])

for(t2 = 0; t2 <= intDiv(T-1,8); t2++) {
  for(t3 = 16*t2; t3 <= min(16*t2+N+13,N+2*T-3); t3++) {
    if (intMod(-N+t3+1,2) == 0 && 16*t2 <= -N+t3+1) {
      s2(N-1);
    }
    for(t4 = max(-16*t2-N+t3+2,0); t4 <= min(-16*t2+2*T-1,-16*t2+t3-1,15); t4++) {
      if (intMod(t4,2) == 0) {
        s2(t3-t4+-16*t2);
      }
      if (intMod(t4+1,2) == 0) {
        s3(t3-t4+-16*t2);
      }
    }
    if (intMod(t3,2) == 0 && 2*T >= t3+2 && 16*t2 >= t3-14) {
      s2(0);
    }
  }
}
```

Figure 15: Three Point Stencil after Time Skewing (for s=8)

4.2 Blocked Storage Mapping

We can achieve scalable locality for the iteration order of Figure 12 by storing the result of iteration $[t, i]$ in `cur[t%2][i]` rather than `cur[t][i]`. This can be done by simply changing the subscript expression for all uses of `cur` in the code in Figure 11, or by supplying the memory mapping $\{ [\ t, i\ ] \rightarrow [\ t \ mod \ 2, i\ ] \}$ to the code generation system described in [SW98].

4.3 Time Skewed Storage Mapping

Both of the above storage mappings will cause a significant number of temporaries to be written to main memory: Each tile writes to a total of $2N$ memory locations, despite the fact that only N live values are produced. We can still use these memory mappings to produce scalable locality, though we must match a quantity that depends on the number of written values, not the number of live values, to the machine balance.

Alternatively, we can employ a memory map that uses a separate small array for all temporary values. Values produced at the end of a block of the time loop (i.e. when $t = s - 1$) are stored in the original array (`cur`). Temporary values are all mapped to a single array that is small enough to fit into cache, and "aligned" with the transformed iteration space (that is, consecutive iterations within a tile write to consecutive locations in the array). This "cache" array must be large enough to hold $2s$ wavefronts plus one additional element, so we simply allocate a 3 by s array, where the first (most significant) index is the wavefront number modulo three, and the second is the position along the wavefront. For our three-point stencil, this gives the memory map

$$[t, i] \rightarrow \begin{array}{ll} \texttt{cur}[\ i\ ], & \text{when } t \ mod \ s = s - 1 \\ \texttt{cache}[\ (t+i) \ mod \ 3, t \ mod \ s\ ], & \text{when } t \ mod \ s < s - 1 \end{array}$$

Given this description of the memory map, the sizes of the arrays, and information about dataflow, the techniques given in [SW98] can be used to check for legality (i.e. ensure that all array references are within bounds and that no live value is overwritten) and generate code.

For stencils of higher dimension, we must also manage the live values that cross between neighboring blocks along the spatial dimension (e.g. the values that flow upward between blocks in Figure 13). We create

additional arrays, which we call "tide" arrays, for each blocked spatial dimension. For our example, there are two live layers at these borders, so we will need tide arrays that are 2 by s_i by N. The storage mapping used is thus

$$
\begin{array}{llll}
[t, i_1, i_2 ... i_n] \rightarrow & \texttt{cur}[\ i_1, i_2 ... i_n\ ], & & \text{when } t \ mod \ s_t = s_t - 1, \\
& \texttt{tide}_j[\ ((i_j + t) \ div \ s_j) \ mod \ 2, & & \text{when } xx_j + 2 \geq s_j \\
& \quad \Sigma_{l=1}^{n} i_l - t, & & \quad \wedge(\nexists k > j \text{ s.t. } xx_k + 2 \geq s_k) \\
& \quad xx_1 - s_1 + 2, ... & & \quad \wedge t \ mod \ s_t \neq s_t - 1, \\
& \quad xx_{n-1} - s_{n-1} + 2, t \ mod \ s_t], & & \\
& \texttt{cache}[\ (t + \Sigma_{l=1}^{n} i_l) \ mod \ 3, xx_1, ... xx_{n-1}, t \ mod \ s_t\ ], & & \text{otherwise,}
\end{array}
$$
where $xx_l = (i_l + t) \ mod \ s_l$

We say that we have time skewed the memory mapping, as well as the iteration space, when we use this memory mapping. Note that the subscript expressions for the `cache` array are identical to the inner loop expressions for the iteration space transformation at the end of Section 3. This ensures that we can always subscript the cache array with the inner loop index variables, ensuring a very simple access pattern with unit stride.

For a stencil of dimension d, the storage mappings of the previous two sections used $2N^d$ memory locations to produce the N^d live values at the end of each time block. In contrast, the number of memory locations used by the time skewed mapping is

$$\underbrace{N^d}_{\texttt{cur}} + \underbrace{(d-1)2sN^{d-1}}_{d-1 \ \texttt{tide} \ \text{arrays}} + \underbrace{3s^d}_{\texttt{cache}}.$$

As N grows larger than s, this approaches N^d rather than $2N^d$, or half the memory traffic of the other storage mappings.

4.4 Optimizing Functional Programs

Note that the blocked or time skewed storage mappings could be applied to a program that was originally written in functional form, such as that shown in Figure 10. Other techniques that transform the iteration space but not the storage mapping cannot produce scalable locality for this code, as the number of addresses touched grows linearly with the number of calculations.

4.5 But What About...

Traditional memory bandwidth optimizations such as loop tiling are generally combined with other transformations, such as array transposition to produce good spatial locality, array padding or copying to prevent cache conflicts [LRW91, TGJ93]. These can also involve a search of possible transformations, and the selection may influence, or be influenced by, the choice of iteration space transformation.

Our time skewed storage mapping lets us ignore these issues. A brief review of the transformed iteration space (Figure 12) makes the reasons for this clear. The `cache` array contains exactly those iterations in the gray wavefront. To advance this wavefront, we begin by reading a new value from the `cur` array. This could cause a cache miss, and possibly collide with part of our `cache` array that is in cache, causing a second miss. However, we then run through s iterations, each of which reads and writes only to the `cache` array, which should (except for the aforementioned single conflict) remain in L1 cache. This eliminates the need for enhancement of spatial locality, or padding or copying to prevent conflicts.

It may still be beneficial combine time skewing with prefetching (to reduce stalls due to latency [MLG92]), especially for systems with extremely high memory balance. We have not investigated doing so at this time.

5 Generalizing Time Skewing

In this section, we generalize our algorithm beyond the simple single-calculation stencils of the previous sections. We use the code from the `TOMCATV` program of the SPEC95 benchmark set (shown in Figure 16) as an example of this algorithm. This code executes up to `T` time steps, each of which performs numerous

```
    do t = 1, T

//    find residuals of iteration t
        rxm(t) = 0.
        rym(t) = 0.
        do j = 2, n - 1
            do i = 2, n - 1
                xx = x(i+1,j) - x(i-1,j)
                yx = y(i+1,j) - y(i-1,j)
                xy = x(i,j+1) - x(i,j-1)
                yy = y(i,j+1) - y(i,j-1)
                a = (xy * xy + yy * yy) * .25
                b = (xx * xx + yx * yx) * .25
                c = (xx * xy + yx * yy) * .125
                aa(i,j) = -b
                dd(i,j) = b + b + a * 2.0408163265306123
                pxx = x(i+1,j) - x(i,j) * 2. + x(i-1,j)
                qxx = y(i+1,j) - y(i,j) * 2. + y(i-1,j)
                pyy = x(i,j+1) - x(i,j) * 2. + x(i,j-1)
                qyy = y(i,j+1) - y(i,j) * 2. + y(i,j-1)
                pxy = x(i+1,j+1) - x(i+1,j-1) - x(i-1,j+1) + x(i-1,j-1)
                qxy = y(i+1,j+1) - y(i+1,j-1) - y(i-1,j+1) + y(i-1,j-1)
                rx(i,j) = a * pxx + b * pyy - c * pxy
                ry(i,j) = a * qxx + b * qyy - c * qxy

//    determine maximum values rxm, rym of residuals
        do j = 2, n - 1
            do i = 2, n - 1
                rxm(t) = max(rxm(t),d_abs(rx(i,j)))
                rym(t) = max(rym(t),d_abs(ry(i,j)))

//    solve tridiagonal systems (aa,dd,aa) in parallel, lu decomposition
        do i = 2, n - 1
            d(i,2) = 1. / dd(i,2)
        do j = 3, n - 1
            do i = 2, n - 1
                r = aa(i,j) * d(i,j-1)
                d(i,j) = 1. / (dd(i,j) - aa(i,j-1) * r)
                rx(i,j) = rx(i,j) - rx(i,j-1) * r
                ry(i,j) = ry(i,j) - ry(i,j-1) * r
        do i = 2, n - 1
            rx(i,n - 1) = rx(i,n - 1) * d(i,n - 1)
            ry(i,n - 1) = ry(i,n - 1) * d(i,n - 1)
        do j = n - 2, 2, -1
            do i = 2, n - 1
                rx(i,j) = (rx(i,j) - aa(i,j) * rx(i,j+1)) * d(i,j)
                ry(i,j) = (ry(i,j) - aa(i,j) * ry(i,j+1)) * d(i,j)

//    add corrections of t iteration
        do j = 2, n - 1
            do i = 2, n - 1
                x(i,j) = x(i,j) + rx(i,j)
                y(i,j) = y(i,j) + ry(i,j)

        if (d_abs(rxm(t)) <= 5e-9 && d_abs(rym(t)) <= 5e-9)      break
```

Figure 16: TOMCATV benchmark from SPEC95

calculations to define new values for the `x` and `y` arrays in terms of `x` and `y` from the previous time step, and records the maximum residual `x` and `y` for each step.

This code defies the techniques from the previous sections for two reasons: First, there may be an early exit from the time step loop (from the last line of Figure 16); Second, the dependences do not follow the same simple patterns: Most notably, the `j` loops in the fourth and sixth loop nests carry dataflow, so we cannot tile them.

When the time step loop is a `while`, or a `for` that contains a `break`, we create an array of boolean values representing the value of the condition in each iteration and convert the statement into an expression that simply computes and saves this value. At the end of each block of time steps, we scan this array to determine if a break occurred in any time step in the time block we have just completed. If it has, we record the number of the iteration in which the break occurred, roll back the calculation to the beginning of the time block, and restart with the upper bound on the time loop set to the iteration of the break.

We can maintain old data for rollback with a simple modification of our memory mapping: We add another dimension to our `cur` array, and replace the

$$[t, i_1, i_2 ... i_n] \rightarrow \texttt{cur}[\ i_1, i_2 ... i_n\ ] \text{ when } t \ mod \ s = s - 1$$

part of the storage mapping with

$$[t, i_1, i_2 ... i_n] \rightarrow \texttt{cur}[\ (t \ div \ s) \ mod \ 2, i_1, i_2 ... i_n\ ] \text{ when } t \ mod \ s = s - 1.$$

Since the $(t \ div \ s) \ mod \ 2$ can be hoisted to the outermost loop, and we already assume that all writes to `cur` will be cache misses, this should not introduce significant computational or memory overhead during the calculation. After the loop, we must either copy from the relevant part of `cur` into an array giving the final result, or (when possible) simply provide a pointer to the appropriate part of `cur`.

The most general way to handle complicated dataflow is with Pugh and Rosser's *iteration space slicing* [Ros98]. This technique follows inter-iteration dependences to find all statement executions that affect (or are affected by) any given target iteration of a given target statement. We use the term *marginal iteration space slicing* to refer to Pugh and Rosser's techniques for finding the additional statement executions that must be added to a slice when a new iteration is added to the target.

For example, consider once again our three point stencil in functional style (Figure 10). The slice required to compute the value for any given iteration follows the dataflow arcs back from that iteration (see Figure 12) forming a triangle (or, in higher dimensions, a pyramid) with the target at its point. For example:

- The slice needed to compute the value in iteration $[3, 1]$ is $\{\ [t, i] \mid t < 3 \wedge t + i \leq 3\ \}$.
- The marginal slice needed to compute $[3, 2]$ given that the slice for $[3, 1]$ is complete is $\{\ [t, i] \mid t < 3 \wedge t + i = 4\ \}$.
- The marginal slice needed to compute $[3, 2]$ given that we have completed the slice for $[3, 1]$ and everything else in the first time block (i.e. $\{\ [t, i] \mid t < 2\ \}$) is $\{\ [t, i] \mid 2 \leq t < 3 \wedge t + i = 4\ \}$.

Iteration space slicing does not require a known target iteration number; we can slice for an iteration identified with variables such as $[t, i]$. For t at the end of a time block, the marginal slice for $[t, i]$, given that we have completed all prior time blocks and the slice for $[t, i-1]$, is $\{\ [t', i'] \mid \leq t - s + 1 \leq t' < t \wedge t' + i' = t + i\ \}$, i.e., exactly the wavefront we use for this example. This is no accident.

The general time skewing algorithm is given in Figure 17. The rationale for each step will become clear as we discuss the application of the algorithm to the stencils we have seen, and a more complex benchmark program.

5.1 The Three Point Stencil Revisited

For the code in Figure 9, Steps 1 - 3 of our general algorithm simply find the dependences shown in Figure 14 and identify that the entire loop nest must be examined to find scalable locality. In Step 4, we may choose to remap storage or not. We consider both options here, as most steps produce the same result.

For Step 5, we choose iteration $[t_e, 1]$ as our extreme point. This defines S_0 as a triangle reaching back along the dependences of Figure 14 (if we don't plan to remap storage) or the dataflow of Figure 12 (if we

1. **Find the array dataflow dependence relations.**
2. **Find the most computationally intensive program regions**, based on symbolic volume computation [Pug94], profiling information, or the depth of loops with non-constant bounds.
3. **Find loop(s) with scalable compute balance**, by searching outward from the most intensive regions. We call this the time step loop, even though it may not actually correspond to such a loop. If none exists, give up; if multiple intensive regions have separate time step loops, optimize each one independently.
4. **Decide whether or not storage remapping will be used.** If the number of memory locations grows more than linearly with the amount of live data (as in the functional style stencil) storage remapping may be required to achieve scalable locality. In other cases, remapping can reduce memory traffic and have other benefits (see Section 6).
5. **Find the volume inside the first full wavefront.** Pick an "extreme point" in the iteration space of some time step, such as the first or last iteration, and compute the backward slice. If doing storage remapping, base this slice on dataflow; otherwise, use all dependences, to create a *movable slice* [Ros98, Section 5.1.3]. We call this the *initial slice*, or S_0.
6. **Find the "slopes" of the initial slice.** The algorithms used to compute dependence differences in [PW92] can be used to find the "slope" of the front surface of the initial slice: For each dimension (loop) i these algorithms produce either a known fixed limit on $\frac{\Delta i}{\Delta t}$ or "infinity" (unbounded). We cannot, in general, tile unbounded directions; their presence indicates that we cannot achieve scalable locality for a fixed cache size. A bounded slope in a direction with an unbounded number of iterations is required for scalable locality; If all directions are unbounded, give up.
7. **Choose an order for traversing iterations** in the final time step, by one of the methods shown in Figure 18. This enumeration produces one or more outer loops over the final values; within them, the iterations in the marginal slice for the corresponding value will be executed by a set of inner loops.

   Slicing can also be used to find an order for executing the iterations within S_0. For example, by executing slices for the extreme point with t_e replaced with each value from 0 to t_e, we would execute the iterations within S_0 with the same wavefront used for the rest of the tile under Option b.
8. **Choose dimensions to be blocked.** Block all but the outermost spatial loop (if legal).
9. **Find balance of tiles and number of temporaries.** The number of temporaries is determined by the number of iterations in each wavefront and the number of wavefronts crossed by any dependence. The balance of tiles is determined by the volume of the tile and the number of live values that cross tile borders. These can be computed via symbolic volume computation [Pug94], or approximated by using dependence distances and the slope of the wavefront.
10. **Choose block sizes.** Set the tile size large enough to produce the compute balance needed for the target architecture. The ratios between the block sizes can be found from the dependence distances (as in Section 3), and the ratio of the sizes to balance from the volume to border ratio for the tile.
11. **Choose a memory mapping**, if doing storage remapping. The storage mapping is defined as follows: • Store values computed at the ends time blocks in the original array (expanded by a factor of two if allowing rollback) • Store values that live past the edge of each block in a "tide" array, whose shape matches that of the block • Store remaining values in the "cache" array, whose shape matches the intersection of the wavefront and the tile.
12. **Generate code.** Given the block sizes and order of enumerating final values, use Pugh and Rosser's techniques to perform marginal slicing and generating code from the results [KPR95, Ros98]. If remapping storage, use extensions of Shen and Wonnacott [SW98].

Figure 17: General Time Skewing Algorithm

There at several ways to choose an order for the final iterations:

a Traverse the dimensions of the original iteration space, putting dimensions with the lowest slope as the outermost loops. This ensures that the dimensions in which more values are reused will be traversed by the innermost loops.

b Traverse according to the wavefront defined by the surface of the initial slice. That is, in step k, perform all iterations I such that $[t-k, I] \in S_0 \wedge \forall_{l<k}[t-l] \notin S_0$. Any iterations not covered by this slicing can be executed in a separate loop nest defined by new initial slice from the remaining space. If this process must be repeated more than a small number of times, give up and use the option above.

c Perform iterative forward and backward slices, as discussed in [Ros98, Section 8.3], for each iteration selected by Option b.

Figure 18: Algorithms for Ordering Final Iterations

do). That is, S_0 is either $[t, w, i] \mid 2t + w + i \leq 2t_e + 2$, or $[t, i] \mid t + i \leq t_e + 1$, respectively, and has slope of 2 or 1.

All options for Step 7 will traverse the iteration space by moving upward through i in this example. If we had performed a stencil using `old[i-2]` and `old[i+2]`, Options b and c would traverse first the odd iterations (which are reached when slicing back from $[t_e, 1]$) and then the even iterations (which are reached when slicing back from a second extreme point selected from the remaining iterations, such as $[t_e, 2]$).

Since there is only one spatial dimension, we block only the time loop in Step 8. The resulting tiles have balance $2s$, as we saw in Section 3, so we create blocks of size $s = \frac{b}{2}$ for a machine with balance b.

At this point, if we are not remapping storage, we can simply generate code to execute S_0, followed by marginal slices for increasing i, producing the iteration ordering used in Figure 14.

To produce our time skewed memory mapping, we produce a cache array that is as wide as the time block (s), and as tall as the number of wavefronts covering any dependence (3).

5.2 The Five Point Stencil Revisited

For the code in Figure 8, Steps 1 - 6 once again follow from our iteration space diagram (Figure 13). As in Figure 13, we consider only the code with memory remapping here. Thus, the slopes in both the i and j dimensions are both 1; we make the arbitrary choice of blocking i rather than j below, for consistency with the figure.

For this program, the differences among the options for Step 7 come into play. Option a would generate the values for the time step t_e by running through all values of i in the block before moving on to the next j, i.e. $[t_e, 1, 1], [t_e, 2, 1], [t_e, 3, 1]$.... (Iteration space slicing would ensure that the appropriately skewed wavefront would be used to traverse the earlier time steps.)

Options b and c also starts by using S_0 to compute $[t_e, 1, 1]$. However, these next moves to iterations that were in the S_0 slice for the previous time step, i.e. $[t_e, 2, 1]$ and $[t_e, 1, 2]$. Since the dependences are uniform, the slices for these two iterations will overlap: they both require $[t_e - 1, 1, 1]$ and $[t_e - 1, 2, 2]$. This overlap occurs whenever the dependences are uniform (though uniform dependences are not required for our technique to work). This effect makes Options b and c intuitively appealing, but all options for this step produce scalable locality if the necessary number of wavefronts fit in cache.

Note that Figure 13 gives a simplified (easier to visualize) version of this full transformation. It actually depicts a hybrid approach in which the wavefront is skewed in both directions (as with Option b), but the blocks are not skewed in the j dimension. Furthermore, it shows $s_t = s_i = 2$ rather than the values $3s_t = 6s_i = 2b$ that are appropriate for this code.

5.3 A More Realistic Example

We now return our attention to the `TOMCATV` program of the SPEC95 benchmark set. Steps 1 - 3 identify the entire loop nest from Figure 16 as the smallest scope with scalable balance. We choose iteration $[t_e, 2, 2]$ of

the statements in the seventh (last) loop nest (which generate the only values that live past the iteration) as our extreme point, and slice backward to define S_0. The dependences carried by the j loops in the fourth and sixth loop nests put all the iterations of all j loops into this slice (e.g. iterations $\{ [t_e, j, 2] \mid 2 \leq j \leq n-2 \}$ of the sixth loop). This unbounded slope in the j dimension prevents us from blocking the j loops, and thus from achieving scalable locality for constant cache size. However, we can still block the i loop (which has slope 1), so we continue.

Option a for Step 7 directs us to place the j loop within the i loop. As we move through the j loop, we reuse all of the values of `rx(i,*)` and `ry(i,*)` (which were in S_0 since they are needed for `x(i,2)` and `y(i,2)`) to update `x(i,*)` and `y(i,*)`. The marginal slices for these later j iterations contains iterations from only the last loop nest, since all iterations of j for previous nests were already in the slice for `x(i,2)` and `y(i,2)`. Option b would produce a similar effect with a slightly more complicated iteration space; Option c differs in that S_0 includes all iterations of j for the final calculations of `x(i,*)` and `y(i,*)` (the the "forward slice" from the S_0 for Option b contains these iterations). However, since Options a and b would have executed these iterations right after S_0, this should not produce any significant advantage.

Step 8 directs us to block the innermost of the two loops, but the dependence prevents this, so we block only the time dimension (as before, let s be the block size). The net effect of the steps so far is essentially to treat this code as a sequence of seven vector operations nested inside the t and i loops, a nesting that is essentially similar to the one for the three point stencil (Figure 12).

Each of the resulting tiles will perform about $68n^2s$ operations (counting `max` and `abs` as one each; we could easily substitute some other cost). It generates and consumes the x and y arrays, about $4n^2$ values total, for a balance of $17s$. We therefore set $s = b/17$. The balance of 17 is large enough that many modern machines can achieve adequate locality within a time step, but future advances must look to inter-time-step optimization for improvement.

5.4 The Domain of Our Algorithm

Our technique requires an exact description of the iteration space and the dataflow information. For structured code, this information can be produced via the Omega Test [PW93] if all loop steps are known constants, and all loop bounds (except the number of iterations in an optional outer `while` loop), conditions tested in if statements, and array subscripts can be expressed as affine functions of the outer loop indices and a set of symbolic constants.

The extensive use of iteration space slicing in our general algorithm makes its domain and speed quite sensitive to the slicing and underlying transitive closure algorithms. It is therefore somewhat difficult to give a precise statement of the domain over which it is exact, or the domain over which it is fast. However, we believe it will be exact and efficient for any program with uniform constant dependences (such as (1,0,1) or (0,1,0) in TOMCATV), which should not present any problem for current slicing algorithms. (More general cases *may* also work well, of course.)

It is also not clear how often this full algorithm will produce results that are better than a collection of ad-hoc techniques designed to produce the same results for cases that arise in practice, such as we describe in [Won99]. It may turn out that that the general algorithm is best seen as a specification of the results of time skewing, rather than the fastest technique to produce this transformation in practice.

6 Empirical Results

We have measured the effects of several variants of the time skewing transformation on the stencils of Figures 9 and 8 and a modified version of the TOMCATV benchmark shown in Figure 16. For each of the stencils, we measured the run times for the original code and time skewed code with the original ("orig"), blocked ("bl"), and time skewed ("tskew") storage mappings. Unless otherwise noted, programs were compiled with g++, using the "-O3 -funroll-loops" options, after the manual hoisting of some loop-invariant `div` and `mod` expressions produced by our code generator. These expressions could, in principle, be hoisted automatically using the approach discussed in [CCC+97]. We also simplified our transformation of the five point stencil by setting $s_i = s_t$. Since traditional loop tiling can be applied to the two inner loops of the five point stencil, we also collected data for a tiled version of this code.

Table 1: Run times for Cache Optimization on 200MHz Pentium

Program	s	T	N	original	Time Skewed orig	bl	tskew
3-Point	500	1200	393216	172	69	80	63
5-Point	16*	268	1086	149	119	92	185*
TOMCATV	12	750	513	702	588	619	

* For the 5-point stencil with time skewed memory mapping s=32; the time for s=16 was 231 s.

Table 2: Run times for Cache Optimization on Sun Ultra 60 Model 2300

Program/compiler	s	T	N	original	Time Skewed orig	bl	tskew
3-Point/g++	500	1200	1048576	263	229	150	162
5-Point/g++	16*	714	1086	177	178	120	308*
TOMCATV/g++	12	750	513	295	196	186	
TOMCATV/f77	12	750	513	186	144	137	

* For the 5-point stencil with time skewed memory mapping s=32; the time for s=16 was 494 s.

As we have not implemented the rollback for `break` statements, we modified the TOMCATV code to remove the early loop exit (this exit is not taken for the supplied test data, so this does not affect the result). Our code generator was not able to produce time skewed code for the time skewed storage mapping for the TOMCATV-like program, so only the original and blocked storage mappings were tested; for these memory mappings, we performed an array transpose to produce unit stride accesses to memory. For the blocked storage mapping, we did not expand temporary arrays (those that carry values only within a time step). We also changed `NMAX` from 513 to 575 to reduce cache interference effects, and to 1096 to allow tests with virtual memory.

To determine whether time skewing is beneficial on current architectures, we ran each code on a 200MHz Pentium workstation and a Sun Ultra 60 Model 2300, using arrays too large to fit in L2 cache. We also manually translated the TOMCATV-like program into Fortran to investigate interactions between time skewing and the optimizations performed by the system f77 compiler on the Ultra 60. The results of these runs are shown in Tables 1 and 2. For the stencil code, the tiles were small enough for the “cache” arrays to largely (or completely) fit in L1 cache; for the TOMCATV-like program, the tiles fit in L1 for the Sun and L2 for the Pentium. In all cases, the time skewed code with blocked storage map was faster than the original code, generally by a significant percentage. The original storage mapping, with its different skewing factor, was never faster than the blocked storage map. The time skewed memory map varied from the fastest option to the slowest (slower than the original code for the five point stencil). The use of traditional tiling on the inner loops of the five point stencil (not shown in the table) did not make a significant change in performance: Run-time was reduced to 144 seconds on the Pentium an to 160 seconds on the UltraSparc. For the TOMCATV-like program, time skewing showed a speedup, even though the intra-time-step compute balance of 17 should have been more than adequate for these architectures. This suggests that other intra-time-step locality optimizations could also have produced an improvement over g++ and f77.

To investigate the value of time skewing for machines with extremely high memory balance, we also ran the codes on the Pentium workstation with arrays that were to large to fit in main memory (the virtual memory machine balance for this system is over 500). Since stalls due to virtual memory are not counted against a process’s time, we measured wall clock time (and cpu utilization) for these runs. In all cases, the system was otherwise idle, so low CPU utilization is directly attributable to a high paging rate. These results for these runs are shown in Table 3. In all cases, the intra-time-step compute balance is dramatically lower than the virtual memory machine balance. The time skewed code is many times faster than the original (or tiled) code, no matter what storage mapping is used, and these speeds can only be produced by

Table 3: Run times (and CPU utilization) for Virtual Memory Optimization on Pentium

Program	s	T	N	original	Time Skewed orig	bl	tskew
3-Point	500	20	4325376	4500 (1.7%)	86 (22%)	90 (27%)	44 (70%)
3-Point	500	1200	4325376	281000 (1.6%)	1240 (62%)	1380 (90%)	845 (83%)
5-Point	100	100	2110	23800 (2.0%)	654 (74%)	390 (76%)	287 (99%)
TOMCATV	256	250	1095	27391 (5.0%)	1014 (98%)	1091 (97%)	

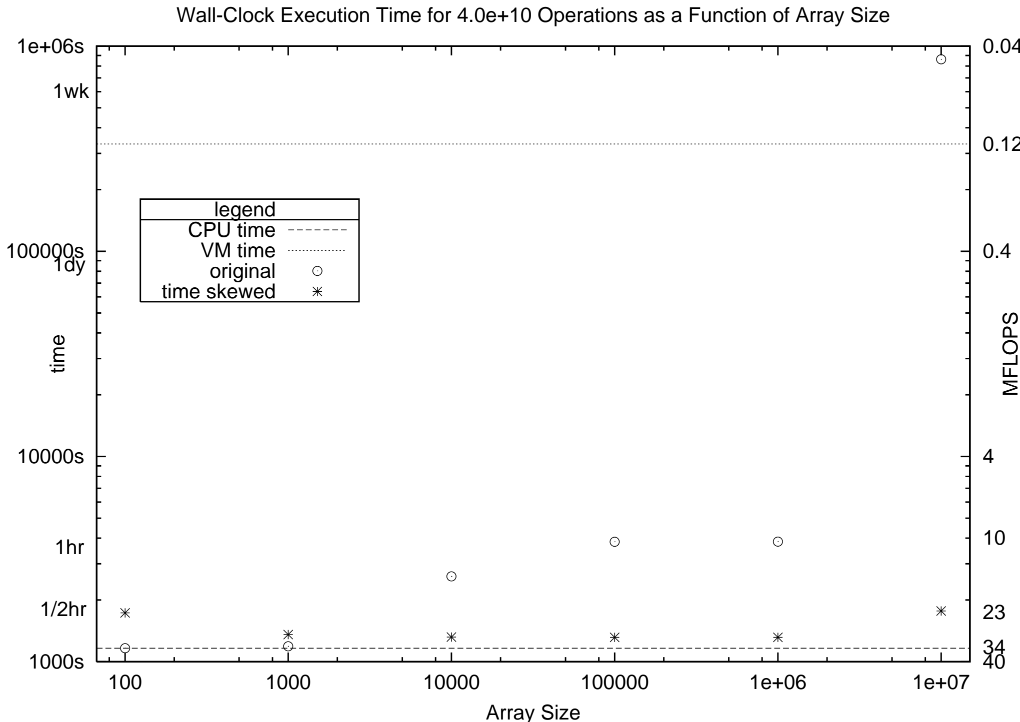

Figure 19: Wall-Clock Execution Time for $4 * 10^{10}$ Operations as a Function of Array Size

inter-time-step locality optimization.

Finally, we performed a series of runs of the three point stencil to compare the original and best time skewed code (i.e. the one with the time skewed memory mapping) for a variety of array sizes. The results are shown in Figure 19. Note that this table shows the time for an additional 10^{10} iterations; it does not include fixed overheads, and T is reduced as N increases.

The dashed "CPU time" line shows the amount of CPU time that would have been required if the CPU had been able to sustain its maximum processing speed (defined as the best speed for the original algorithm on a data set that fits entirely in L1 cache). The circles show the unoptimized run times, which grow from the optimal 20 minutes, to over an hour when the data set does not fit in L2 cache, and then to about 10 days when the data set exceeds main memory size. The stars show the run time of the time skewed version, which keeps a nearly constant time of about 22 minutes, growing close to 30 minutes only for virtual memory and when very small N makes the increased loop overhead significant.

The dotted "VM time" line at about 4 days gives the time the virtual memory system would require, if running continuously at full bandwidth, to transfer the $2 * 10^{10}$ values needed for 10^{10} iterations. This thus represents a lower bound on the time for any run in which a large data set does not have significant inter-time-step locality, regardless of the amount of optimization done to improve locality within a time step or introduce prefetching. We believe that the reduction in overhead from at least 18800% (based on this

lower bound) to about 50% demonstrates that time skewing can be successful at countering extremely high machine balance. It may be the case that prefetching operations could further reduce this overhead, but we have not tested this.

We could add a "main memory time" line on the graph, corresponding to the "VM time" but showing the time to transfer $2 * 10^{10}$ values to main memory at full bandwidth. However, this line would lie off the bottom of the current range, at about 8 minutes, indicating that the original code is slowed down by factors other than simple bandwidth when the array does not fit in L2 cache. This slowdown could be caused by a lack of prefetching or by cache interference. In other cases, a lack of adequate spatial locality could also contribute to a memory bottleneck beyond what was predicted by a comparison of compute balance and machine balance. It is interesting to note that the time skewed code does not seem to suffer from any of these effects (at least for main memory), as we predicted in Section 4.5.

We have not performed experiments to measure the cost of rollback when a `break` causes early termination of a loop, or to measure the cost of allowing rollback when it does not occur. We have also omitted any study of the possibility of further speeding up the time skewed code by performing subsequent optimizations such as loop unrolling. Finally, we have not studied a wide variety of benchmarks to determine how often it is necessary to go beyond the traditional approach of loop fusion followed by tiling of perfect nests. These topics have been covered in detail in recent work by Song and Li [SL99].

7 Related Work

Our approach to studying locality differs most dramatically from other work in that it focuses on answering different questions. Other work asks *"Given a loop nest in a certain domain (e.g. perfectly nested loops), what can be done to produce the best locality?"* We try to answer the questions *"Is the locality of a given program limited to some constant, or might it be made arbitrarily high?"* and, if there is no fixed limit, *"How can we transform the code to achieve arbitrarily high locality?"*

We do not know of any other techniques for identifying scalable locality, or even for quantifying the maximum possible locality for a piece of code or (conversely) the minimum scale on which code may exhibit a certain locality. However, there is a great deal of work that reduces memory bandwidth requirements, producing both small-scale locality improvements with low overhead (to give better cache performance), and larger-scale reductions in I/O for out-of-core data sets. The latter work generally involves replacing data sets that could have been in virtual memory with explicit disk I/O to maintain a subset of the data in main memory, and is based on very different techniques, with overhead that makes it unsuitable for cache optimization. We therefore focus primarily on other cache optimization work.

Most current techniques for improving locality [GJ88, WL91, Wol92, MCT96] are based on the search for groups of references that may refer to the same cache line, assuming that each value is stored in the address used in the original program. They then apply a sequence of transformations to try to bring together references to the same address. However, their transformation systems are not powerful enough to perform the time skewing transformation: the limits of the system used by Wolf and Lam are given in Section 2.7 of [Wol92]; McKinley, Carr, and Tseng did not apply loop skewing, on the grounds that Wolf and Lam did not find it to be useful in practice. Thus, these transformation systems may all be limited by the bandwidth of the loops they are able to transform. For example, without the time loop, none of the inner loops in `TOMCATV` or our stencils exhibits scalable locality. Thus, there are limits to the locality produced by any transformation of the body of the time loop. When a set of perfectly nested loops exhibits scalable balance, as in matrix multiplication or the in-place variants of our stencil calculations, these techniques can produce scalable locality. We have no reason to believe we can produce better results for such codes.

Recent work by Pugh and Rosser [Ros98] uses 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. For example, they can produce a version of `TOMCATV` in which each time step performs a single scan through each array, rather than the five different scans in the original code. However, their system transforms the body of the time loop, without reordering the iterations of the time loop itself, and is thus limited by the finite balance of the calculation in the loop body. Pugh and Rosser also do not discuss the general problem of how to enumerate the elements of a multi-dimensional iteration space, leaving this topic for "future work"

[Ros98, Section 8.4.2].

Work on tolerating memory latency, such as that by Mowry et al. [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, in those cases where latency is so high that even the occasional miss introduces noticeable overhead (e.g. for virtual memory).

We have recently learned of ongoing work by Song and Li [SL99] that also uses a combination of loop skewing and tiling, and a memory expansion that is generally equivalent to our "blocked" layout, to improve locality. Their iteration space transformation is slightly different in that it generates innermost loops are blocks of the original loop, which typically gives good spatial locality without requiring our storage transformations. Song and Li also use speculative execution to handle outer `while` loops. They give a more thorough treatment of practical issues having to do with producing the most efficient resulting code, and give experimental results over a wider set of benchmarks. Our work explores a wider range of storage transformations, including transformations that produce asymptotically minimal memory traffic and reduce the need for other optimizations to address latency, prevent cache conflicts, and create spatial locality. Our storage transformations also let us optimize code that is presented in functional style. Song and Li do not discuss the use of their transformation for systems with extremely high balance.

Our earlier work on time skewing [MW99, Won99] contains some details not discussed here, such as ad-hoc techniques for handling code other than stencils (these are subsumed by our general algorithm in Section 5). We first explored the use of time skewing for out-of-core data sets in [SSW97]. Details of our extensions to the Omega Library's code generation system are given in [SW98]. We have also explored the generalization of time skewing for multiprocessors [Won00b], but these details are outside the scope of this paper.

8 Conclusions

As processors continue to out-pace memory systems, compilers must produce ever higher degrees of locality to ensure efficient processor utilization. This raises the question of whether it is possible to increase locality arbitrarily high in a significant number of programs. Producing extremely high degrees of locality may require unusual combinations of obscure transformations, and thus the inability of current compilers to obtain a given degree of locality is no guarantee that it does not exist in a given code.

We classify calculations according to whether or not they exhibit *scalable compute balance*, which is a necessary (though not sufficient) condition for *scalable locality*. For some such calculations, such as matrix multiplication, we can achieve scalable locality via well-understood transformations such as loop tiling. However, most current techniques for locality optimization do not provide scalable locality for time-step calculations.

The time skewing iteration space transformation can be used to reorder the iterations of a class of time step calculations to permit scalable locality. For many such calculations, this iteration space transformation is sufficient to produce scalable locality (though not necessarily minimal main memory traffic); for others (such as those presented in functional form), storage mapping transformations may be required as well. The time skewing storage mapping produces memory traffic that asymptotically approaches the number of live values that flow between tiles in a time skewed iteration space, and reduces the need for other optimizations to address latency, prevent cache conflicts, and create spatial locality. Time skewing can be successfully used to counter bandwidth limitations for memory systems as different as L2 cache, main memory and virtual memory, bridging gap between cache optimization and out-of-core data processing.

Time skewing can also be used to predict cache requirements for a given program running on a machine with a given machine balance b. In general, N dimensional time-step stencil calculations will require a cache size that grows with b^N.

Time skewing is, in a sense, an extension of the philisophy used for register allocation to the domain of cache. Instead of relying on the programmer's mapping of values to memory, we distill out the fundamental dataflow of the algorithm, transform it to reduce the number of live temporaries, and generate a new storage mapping.

9 Acknowledgements

John McCalpin, Tina Shen, and Jaime Spacco all contributed to the development of the time skewing transformation [MW99, SSW97]. This work was supported by funds from Haverford College and by NSF grant CCR-9808694.

References

[CCC+97] Rohit Chandra, Ding-Kai Chen, Robert Cox, Dror E. Maydan, Nenad Nedeljkovic, and Jennifer M. Anderson. Data distribution support on distributed shared memory multiprocessors. In *ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 334–345, June 1997.

[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.

[EHLP91] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of 4 Perfect benchmark programs. In *Proceedings of the 4th Workshop on Programming Languages and Compilers for Parallel Computing*, August 1991. Also Technical Report 1193, CSRD, Univ. of Illinois.

[EHP98] R. Eigenmann, J. Hoeflinger, and D. Padua. On the automatic parallelization of the Perfect benchmarks. *IEEE Trans. Parallel Distributed Systems*, 9(1), January 1998. Also Technical Report 1392, CSRD, Univ. of Illinois.

[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.

[LRW91] M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. *Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[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.

[Pug94] William Pugh. Counting solutions to presburger formulas: How and why. In *SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[PW92] William Pugh and David Wonnacott. Eliminating false data dependences using the Omega test. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 140–151, San Francisco, California, June 1992.

[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/.

[RMCKB97] Gerald Roth, John Mellor-Crummey, Ken Kennedy, and R. Gregg Brickner. Compiling stencils in high performance fortran. In *Proceedings of SC '97: High Performance Networking and Computing*, November 1997.

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

[Sed98] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley, third edition, 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*, pages 215–228, May 1999.

[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.

[TGJ93] O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing'93*, November 1993.

[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.

[Wol89] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, Mass., 1989.

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

[Won95] David G. Wonnacott. *Constraint-Based Array Dependence Analysis*. PhD thesis, Dept. of Computer Science, The University of Maryland, August 1995. Available as ftp://ftp.cs.umd.edu/pub/omega/davewThesis/davewThesis.ps.

[Won99] 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.

[Won00a] David Wonnacott. Extending scalar optimizations for arrays. In *Languages and Compilers for Parallel Computing*, volume 2017 of *Lecture Notes in Computer Science*, pages 97 – 111. Springer-Verlag, August 2000.

[Won00b] David Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *Proceedings of the 2000 International Parallel and Distributed Processing Symposium*, May 2000.