Rutgers University Department of Computer Science Technical Report 378 Submitted 12 Feb 1999

Achieving Scalable Locality With Time Skewing *

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

Revised to October 9, 1998

Abstract

The widening gap between processor speed and main memory speed has generated interest in compile-time optimizations to improve memory locality. The possibility that processors will continue to outpace memory systems raises the question of whether these techniques can be used to produce ever higher degrees of locality. In this article, we discuss techniques that can be used to produce "scalable locality": locality that can be adjusted for any ratio of processor and memory speeds. We identify a class of calculations for which existing techniques do not generally produce scalable locality, and give an algorithm for obtaining scalable locality for a subset of this class. We include empirical evidence that this transformation can provide dramatic speedups when processor speed is far higher than memory speed.

1 Introduction

The widening gap between processor speed and main memory speed has generated interest in compile-time optimizations to improve memory locality (the degree to which values are reused while still in cache [WL91]). A number of techniques have been developed to improve the locality of "scientific programs" (programs that use loops to traverse large arrays of data) [GJ88, WL91, Wol92, MCT96, Ros98]. These techniques have generally been successful in achieving good performance on modern architectures. However, the possibility that processors will continue to outpace memory systems raises the question of whether these techniques can be "scaled" to produce ever higher degrees of locality.

We say a calculation exhibits *scalable locality* if its locality can be made to grow at least linearly with the problem size while using cache memory that grows less than linearly with problem size. In this article, we show that some calculations cannot exhibit scalable locality, while others can (typically these require tiling). We then discuss the use of compile-time optimizations to produce scalable locality. We identify a class of calculations for which existing techniques do not generally produce scalable locality, and give an algorithm for obtaining scalable locality for a subset of this class. Our techniques make use of *value-based dependence relations* [PW93, Won95, PW98], which provide information about the flow of values in individual array elements among the iterations of a calculation. We initially ignore issues of cache interference and of spatial locality, and return to address these issues later.

We define the balance of a calculation (or *compute balance*) as the ratio of operations performed to the total number of values involved in the calculation that are live at the start or end. This ratio measures the

*This is supported by NSF grant CCR-9808694

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

Figure 1: Three-Point Stencil in Single-Assignment Form

```
// 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

degree to which values are reused by a calculation, and thus plays a role in determining the locality of the running code. It is similar to McCalpin's definition of *machine balance* as the ratio of a processor's sustained floating point operation rate to a memory system's sustained rate of transferring floating point numbers [McC95].

In some cases, limits on the total numbers of operations performed and values produced place absolute limits on compute balance, and therefore on the locality that could be achieved if that code were run in isolation. For example, if the entire array `A` is live at the end of the loop nest shown in Figure 1, the balance of this nest is approximately 3 (N values are live at entry, and $N * T$ are live at exit, after $3 * N * T$ operations). If all the values produced had to be written to main memory, and all of the values that are live come from main memory, then we must generate one unit of memory traffic (one floating point value read or written) for every three calculations performed. For other codes, such as matrix multiplication (Figure 2), the balance grows with the problem size. Thus, for large matrices, we may in principle achieve very high cache hit rates.

Note that compute balance depends on information about what values are live. For example, if only `A[T][*]` is live at the end of Figure 1, then the balance of this code is $\frac{3}{2}T$. This raises the hope that we can achieve scalable locality if we do not store the other values in main memory. Compute balance also depends on the scope of the calculation we are considering. If all elements of `A` are killed in a second loop nest that follows the code in Figure 1, and that nest produces only N values, the balance of these two nests could be higher than the balance of Figure 1 alone. Once again, this raises the hope of improving locality, this time by keeping values in cache between the two nests.

One way to achieve locality proportional to compute balance would be to require a fully associative cache large enough to hold all the intermediate values generated during a calculation. Our definition of scalable locality explicitly rules out this approach.

To achieve scalable locality, we must divide the calculation into an ordered sequence of "stripes" such that (a) executing the calculation stripe-by-stripe produces the same result, (b) each stripe has balance proportional to the problem size, (c) the calculations in each stripe can be executed in an order such that the number of temporary values that are simultaneously live is "small". A value is considered temporary if its lifetime is contained within the stripe, or if it is live on entry to the entire calculation but all uses are within the stripe. By "small", we mean to capture the idea of data that will fit in cache, without referring to

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

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

```
for (int t = 0; t<T; t++)
  for (int i = 1; i<N-1; i++)
    cur[i] = 1.0/3 * (old[i-1] + old[i] + old[i+1]);
  if (t < T-1)
    for (int i = 0; i<N; i++)
      old[i] = cur[i];
```

Figure 4: Time-Step Three Point Stencil

any particular architecture. In particular, we wish to avoid cache requirements that grow linearly (or worse) with the size of the problem. In some cases, such as the simple stencil calculations discussed in [MW98] we can limit these sizes to functions of the machine balance; in other cases, such as the `TOMCATV` benchmark discussed in Section 4, the cache requirement grows sublinearly with the problem size (for `TOMCATV` it grows with the square root of the size of the input, as well as with the balance).

Existing techniques can produce scalable locality on some codes. For example, tiling matrix multiplication [WL91] produces scalable locality. Figure 3 shows the resulting code, for tile size s. In our terminology, the `jb` and `kb` loops enumerate s^2 stripes, each of which executes n tiles of size s^2, and has a balance of $\frac{2ns^2}{2ns+s^2}$. Within each stripe, the total number of temporaries that are live simultaneously does not exceed $s^2 + 2s$ (for one tile of `B` and one column of a tile of `A` and `C`). Thus, by increasing s to match the machine balance, we could achieve the appropriate locality using a cache of size $O(s^2)$ (ignoring cache interference). However, there are calculations for which current techniques cannot produce scalable locality, as we will see in the next section.

The remainder of this paper is devoted to a discussion of achieving scalable locality for a class of calculations we call *time-step calculations*. Section 2 defines this class of calculations, and shows how to produce scalable locality for a simple example via *time skewing* [MW98]. Section 3 generalizes time skewing beyond the limited class of problems discussed in [MW98]. Section 4 presents empirical studies on benchmark codes. Section 5 discusses other techniques for improving locality, and Section 6 gives conclusions.

2 Time-Step Calculations and 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, the three point stencil calculation in Figure 4 is a time-step calculation. It computes a new value of

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

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

`cur[i]` from the values of `cur[i-1..i+1]` in the previous iteration of `t`. This flow of values is essentially the same as that shown in Figure 1. In contrast, the value computed in iteration $[t, i]$ of the "in-place" stencil shown in Figure 5 is used in iteration $[t, i+1]$ (as `cur[i-1]`), so we do not call this loop nest a time-step calculation.

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 calculations (by producing stripes that combine several time steps).

The techniques presented by Wolf and Lam [WL91, Wol92] can be used to achieve scalable locality for Figure 5, but they cannot be applied to calculations with several loop nests, such as the time-step calculation in Figure 4. In [MW98], we describe the time skewing transformation, which can be used to achieve scalable locality for Figure 4. As originally formulated, this transformation could only be applied to time-step stencil calculations (in which each array element is updated using a combination of the element's neighbors), and only if the stencil has only one statement that performs a calculation (statements that simply move values, such as the second assignment in Figure 4, are allowed). In the next section, we describe a more general form of time skewing. The remainder of this section reviews the original formulation of time skewing, as it applies to Figure 4.

The essential insight into understanding time skewing is that it applies a fairly conventional combination of skewing and tiling to the set of dependences that represent the flow of values (rather than memory aliasing). For example, in Figure 4, the value produced in iteration $[t, i]$ of the calculation is used in iterations $[t+1, i-1]$, $[t+1, i]$, and $[t+1, i+1]$. If we had a single loop nest with this dependence pattern, the algorithm of Wolf and Lam would skew the inner loop with respect to the time loop, producing a fully permutable nest. It would then tile this loop nest to achieve the appropriate degree of locality.

We can perform this skewing and tiling if we first expand the `cur` array and forward substitute the value of `old`. In the resulting code, there are $O(B)$ values produced in each tile for consumption in the next tile (where B is the tile size). A stripe of N such tiles produces $O(N)$ values while performing $O(N * B)$ operations on $O(N * B)$ temporaries, $O(B)$ of which are live simultaneously. Thus, we may hope to achieve scalable locality. Unfortunately, expanding the `cur` array causes each temporary to be placed in a unique memory location, which does not yield an improvement in memory locality.

To improve locality, we must either recompress the expanded array in a manner that is compatible with the new order of execution, or perform the skewing and tiling on the original imperfect loop nest (this requires a combination of unimodular and non-unimodular transformations). Both of these approaches are discussed in detail in [MW98]. In the code that results from the first, all iterations except those on the borders between tiles do all their work with a single array that is small enough to fit in cache. This array should reside entirely in cache during these iterations, allowing us to ignore issues of spatial locality and cache interference. Furthermore, this technique lets us optimize code that is originally presented in single-assignment ("functional") form, like Figure 1. However, this approach taxes the code generation system that we use to its limits, even for single-statement stencils. This causes extremely long compile times and produces code with a great deal of additional integer math overhead due to the loop structure [SW98].

3 A General Algorithm for Time Skewing

In this section, we present a more general algorithm for time skewing time-step calculations, using as an example the code from the `TOMCATV` program of the SPEC95 benchmark set (shown in Figure 6). We begin by giving the domain of our algorithm and our techniques to coerce aberrant programs into this domain, and then present the algorithm itself.

3.1 The Domain of Our Algorithm

As with any time-step calculation, all loop-carried dataflow must come from the previous iteration of the time loop (in `TOMCATV`, the `t` loop). Note that the `j` and `i` loops of the second nest (the "determine maximum" nest) carry reduction dependences [Won95], and do not inhibit time skewing.

The `j` loops in the fourth and sixth nests (the two dimensional loops under "solve tridiagonal") do carry dataflow, so these at first appear to prevent application of time skewing. However, we can proceed with the algorithm if we treat each column of each array as a single vector value, and do not attempt to block this dimension of the iteration space (this will have consequences for our cache requirements, as we shall see below). It may be possible to extend our algorithm to handle cases in which there is a loop that carries a dependence in only one direction along such a vector, but we have not investigated this possibility. In this case, the fourth nest carries information forward through the `j` dimension, and the sixth carries it backward along this dimension, which rules out skewing in this dimension.

Our algorithm is restricted to the subset of time-step calculations that meet the following criteria.

3.1.1 Affine control flow

All loop steps must be known, and all loop bounds and all conditions tested in if statements must be affine functions of the outer loop indices and a set of symbolic constants. This makes it possible to describe the iteration spaces with a set of affine constraints on integer variables, which is necessary because we use the Omega Library [KMP+95] to represent and transform these spaces. We do allow one exception to this rule, however: conditions controlling the execution of the outer loop need not be affine. Such conditions may occur due to breaks, while loops, or simply very complicated loop bounds. These are all handled in the same way, though we present our discussion in terms of break statements, since this is what occurs in `TOMCATV`.

3.1.2 Uniform loop depth and restricted intra-iteration dataflow

Every statement within the time loop must be nested within the same number of loops, and the flow of information within an iteration of the time loop must connect identical indices of all loops surrounding the definition and use. For example, consider the value produced by the last statement in the first nest ("find residuals"). The value produced in iteration $[t, j, i]$ (and stored in `ry(i,j)`) is used in iteration $[t, j, i]$ of the fourth nest. Note that the reference to `ry(i,j-1)` in this statement does not cause trouble because we have already given up on skewing in the `j` dimension.

In some cases we may be able to convert programs into the proper form by simply reindexing the iteration space. For example, if the first `i` loop ran from `1` to `n-2` and produced `ry(i+1,j)`, we could simply bump the `i` loop by 1. If the calculation involves nests of different depths, we add single-iteration loops around the shallower statements. The third loop nest of `TOMCATV` (the first in the "solve tridiagonal" set) has only two dimensions, so we add an additional `j` loop from `2` to `2` around the `i` loop.

We perform this reindexing by working backwards from the values that are live at the end of an iteration of the time loop (in this example, `rxm(t)` and `rym(t)`, which are used later, and `x(*,*)` and `y(*,*)`, which

```
    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 6: `TOMCATV` benchmark from SPEC95

are used in iteration $t+1$). We tag the loops containing the writes that produce these values as "fixed", and follow the dataflow dependences back to their source iterations (for example, the dataflow to iteration $[t, j, i]$ of the write to `y` in the last nest comes from iteration $[t, j, i]$ of the sixth nest when $j < n-1$, and $[t, i]$ of the fifth nest when $j = n-1$. We then adjust the iteration spaces of the loops we reach in this way, and "fix" them. The sixth nest is simply fixed, and the fifth has a `j` loop from `n-1` to `n-1` wrapped around it (since iteration $[t, n-1, i]$ reads this value). If we ever need to adjust a fixed loop, the algorithm fails (at least in one dimension). We then follow the dataflow from the statements we reached in this nest, and so on, until we have explored all dataflow arcs that do not cross iterations of the time loop. Any loops that are not reached by this process are dead and may be omitted.

3.1.3 Finite inter-iteration dataflow dependence distances

For our code generation system to work, we must know the factor by which we will skew. This means that we must be able to put known (non-symbolic) upper and lower bounds on the difference between inner loop indices for each time-loop-carried flow of values. For example, the first loop nest of `TOMCATV` reads `x(i+1,j)`, which was produced in iteration $[t-1, i+1, j]$, so our upper bound on the difference in the `i` dimension must be at least 1. We cannot apply our current algorithm to code with coupled dependences (e.g., if the first nest read `x(i+j,j)`).

3.2 Time Skewing

Consider a calculation from the domain described above, or a calculation such as `TOMCATV` for which certain dimensions are within this domain. All values used in an iteration of the time loop come from within a fixed distance $\pm\delta_l$ of the same iteration of loop l in the previous time step. Therefore, the flow of information does not interfere with tiling if we first skew each loop by a factor of δ_l. This is the same observation we made for Figure 4 in Section 2. In fact, if we think if all the `j` dimensions of all seven arrays as a single 7 by N matrix value, the dataflow for `TOMCATV` is *identical* to that of Figure 4.

We therefore proceed to skew and tile the loops as described above. This requires that we fuse various loop nests that may not have the same size. Fortunately, this is relatively straightforward with the code generation system [KPR95] of the Omega Library. We simply need to provide a linear mapping from the old iteration spaces to the new. The library will then generate code to traverse these iteration spaces in lexicographical order. Given g loops $l_1, l_2, ...l_g$ that are within our domain, and e loops $h_1, h_2, ...h_e$ that are not, we produce the iteration space

$$\begin{array}{l}[\ t/B, (l_1+\delta_{l1}*t)/B, (l_2+\delta_{l2}*t)/B, ..., (l_{g-1}+\delta_{l_g-1}*t)/B, l_g+\delta_{l_g}*t,\\ \ \ (l_{g-1}+\delta_{l_g-1}*t)\%B, (l_2+\delta_{l2}*t)\%B, ..., (l_1+\delta_{l1}*t)\%B, t\%B,\\ \ \ nn, h_1, h_2, ..., h_{e-1}\ ]\end{array}$$

where nn is the number of the nest in the original ordering, and B is the size of the tiles we wish to produce. This is the same as the formulation given in [MW98], with the addition of the constant levels and levels not within the domain.

For `TOMCATV`, we transform the original set of iteration spaces from $[\ t, nn, j, i\ ]$ to $[\ t/B, i{+}t, t\%B, nn, j{+}t\ ]$, where nn ranges from 1 to 8 (the initialization of `rxm` and `rym` counts as a nest of 0 loops).

The resulting g outer loops traverse a set of stripes, each of which contains $T * B^g$ iterations that run each statement through all the e loops. The $g+e$ inner loops that constitute a tile perform $O(B^g * E)$ operations on $O(B^g * E)$ floating point values, where E is the total size of the "matrix" that constitutes the value produced by the e loops executing all for statements. All but $O(B^{g-1} * E)$ values are consumed by the next tile, providing stripes with $O(B)$ balance, and, as long as E grows less than linearly with the size of the

problem, the hope of scalable locality. (In `TOMCATV`, E is a set of seven arrays of size N, while the problem involves arrays of size N^2, so our cache requirement will grow with the square root of the problem size).

As at the end of Section 2, we are left with the question of how to store these values without either corrupting the result of the calculation (e.g. if we use the original storage layout) or writing temporaries to main memory in sufficient quantities to inhibit scalable locality (e.g. by fully expanding all arrays). In principle, if we know E, we could apply the layout algorithm given in [MW98], producing an array of temporaries that will fit entirely in cache. It may even be possible to develop an algorithm to perform this operation with B and E as symbolic parameters. However, in the absence of major improvements to the implementation of the code generation systems of [KPR95] and [SW98], this method is impractical.

Instead, we simply expand each array by a factor of two, and use $t\%2$ as the subscript in this new dimension. This causes some temporaries to be written out to main memory, but the number is proportional to the number of non-temporary values created, not the number of operations performed, so this does not inhibit scalable locality. This method also forces us to contend with cache interference (which we simply ignore at this point, though we could presumably apply algorithms for reducing interference to the code we generate). Finally, our code may or may not traverse memory with unit stride. If possible, the arrays should be transposed so that a dimension corresponding to the innermost loop scans consecutive memory locations.

3.2.1 Break statements

We apply our algorithm to a calculation involving a break that is guarded with a non-affine condition as follows: 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 iteration of the outer loop (which steps through blocks of iterations in the original time loop), 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 preserve the data that are present at the end of each time block by using two arrays for the values that are live at the ends of time blocks (one for even blocks, and one for odd blocks). This can double the total memory usage, but will not affect the balance or locality of the calculation (except to the degree that it changes interference effects).

If it is possible to determine that a break does not affect the correctness of the result, we can avoid the overhead of the above scheme by simply stopping the calculation at the end of the time block in which the break occurred. Unfortunately, we know of no way to determine the purpose of a break statement without input from the programmer (possibly in the form of machine-readable comments within the program itself).

4 Empirical Results

NOTE TO REVIEWERS: At this time, I only have results for the TOMCATV benchmark running with virtual memory. For the final version, I also expect to have results on one or more workstations, such as a Sun Ultra/60 and several SGI machines, and include a larger set of benchmarks. Based on experiences with stencil calculations, I expect that these machines will show either a small gain or a small loss in performance, but that getting peak performance will require manually hoisting some loop-invariant expressions (or getting a better complier...).

To verify the value of time skewing in compensating for extremely high machine balance, we tested it using the virtual memory of a Dell 200MHz Pentium system running Linux. This system has 64 M of main

memory, 128K of L2 cache, and 300M of virtual memory paged to a swap partition on a SCSI disk. This test was designed to test the value of time skewing on a system with extremely high balance.

We transformed the `TOMCATV` benchmark according to the algorithm given in the previous section, except for the break statement (this break is not taken during execution with the sample data). We also increased the array sizes from 513 by 513 to 1340 by 1340, to ensure that all seven arrays could not fit into main memory (the seven arrays together use about 96 Megabytes).

The original code required over 9 minutes per time step (completing a run with T=8 in 4500 seconds, and a run with T=12 in 6900 seconds). For the time skewed code, we increased T to 192 to allow for a sufficiently large block size. This code required under 20 seconds per time step (completing all 192 iterations in 3500 seconds). Thus, for long runs, performance was improved by more than a factor of 30.

The loop nests produced by the time skewing transformation may be more complicated than the original loops, so the transformed code may be slower than the original for small problems. For example, when the original `TOMCATV` data set is used, the entire data set fits in main memory, but any tile size greater than 2 exceeds the size of the L2 cache. In this case, the time skewed code is slower by a factor of two.

5 Related 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 (unoptimized) 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` exhibits scalable locality. Thus, there are limits to the locality produced by any transformation of the body of the time loop.

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.

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

6 Conclusions

For some calculations, such as matrix multiplication, we can achieve scalable locality via well-understood transformations such as loop tiling. This means that we should be able to obtain good performance for these calculations on computers with extremely high machine balance, as long as we can increase the tile

size to provide matching compute balance. However, current techniques for locality optimization cannot, in general, provide scalable locality for time-step calculations.

The time skewing transformation described here can be used to produce scalable locality for many such calculations, though it increases the complexity of the loop bounds and subscript expressions. For systems with extremely high balance, locality issues dominate, and time skewing can provide significant performance improvements. For example, we obtained a speedup of a factor of 30 when running the `TOMCATV` benchmark with arrays that required virtual memory.

References

[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.

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

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

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

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

[MW98] John McCalpin and David Wonnacott. Time skewing: A value-based approach to optimizing for memory locality. 1998. In preparation. A preprint is available as http://www.haverford.edu/cmsc/davew/cache-opt/tskew.ps.

[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.

[PW98] William Pugh and David Wonnacott. Constraint-based array dependence analysis. *ACM Trans. on Programming Languages and Systems*, 20, may 1998. Available as http://www.haverford.edu/cmsc/davew/array-analysis/WhatHowWhy.ps.

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

[SW98] Tina Shen and David Wonnacott. Code generation for memory mappings. 1998. In preparation. A preprint is available as http://www.haverford.edu/cmsc/davew/cache-opt/mmap.ps, and an earlier version of this work appeared in the 1998 Mid-Atlantic Student Workshop on Programming Languages and Systems (MASPLAS '98).

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

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

[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.