Rutgers University Department of Computer Science Technical Report 379
Submitted 2 March 1999

Time Skewing: A Value-Based Approach to Optimizing for Memory Locality*

John McCalpin
Silicon Graphics Computer Systems
2011 N. Shoreline Blvd,
Mountain View, CA 94043.
email: mccalpin@sgi.com

David Wonnacott
Department of Computer Science
Haverford College
Haverford PA 19041.
email: davew@cs.haverford.edu

Revised to September 10, 1998

Abstract

As the gap between processor and main memory speed continues to grow, higher cache hit rates are required for efficient processor use. Recent work on compile-time transformations to improve locality in scientific progams has focused on loop fusion, tiling, and distribution; previous work suggests that loop skewing is not useful in optimizing for locality. In this article, we show that the value of loop skewing may only be evident in a compiler that includes transformations that have not been applied in empirical studies of locality (such as the interchange of imperfectly nested loops). We also show how a new approach to data transformation can be used to further reduce memory traffic for these calculations.

1 Introduction

Since about 1990, the relative cost (in time) of main memory accesses and floating-point arithmetic have undergone a dramatic shift for microprocessor-based computers. One particular metric is the time required to load a word from a unit-stride data stream relative to the time required to perform a floating-point operation. As shown in [McC95], this ratio has gone from near 1 in 1990 to at

*This work is supported by funds from Haverford College.

least 20 in 1996, with an annual increase near 70% per year averaged across the industry.

This increase in the cost of memory accesses relative to floating-point arithmetic has had different impacts on different application areas, with some algorithms in computational fluid dynamics and signal processing/image analysis showing the largest sensitivity to the cost of memory accesses.

In this article, we discuss techniques for improving memory locality of simple "stencil operations", which are prototypical for explicit algorithms in computational fluid dynamics and signal processing/image analysis. These algorithms iteratively apply a nearest-neighbor operator to evolve an initial condition to a final state.

Wolf and Lam showed that, for some stencil calculations, loop skewing plays a crucial role in improving memory locality [WL91, Figures 2 and 7], [Wol92, Figure 4.1]. However, their studies of compile-time locality optimizations on benchmark codes [Wol92] suggest that loop skewing does not play a significant role in improving locality in practice. Some recent approaches to locality optimization have overlooked loop skewing (for example, [MCT96] and [RMCKB97]), on the grounds that Wolf and Lam found it was not useful.

In Section 2 of this article, we give example stencil calculations that are outside the domain of the techniques of Wolf and Lam, and show that loop skewing and tiling can still be used to improve locality. This leads us to conclude that loop skewing should not be overlooked as a useful transformation for improving memory locality in compilers that can handle more general loop transformations. For our examples, loop skewing must be combined with either (a) interchange of imperfectly nested loops, or (b) forward substitution of an array expression and a new form of data transformation. The latter form produces less traffic to main memory, but at the cost of a significant increase the complexity of loop bounds and subscript operations. We give the details of this combination of iteration space and data transformation, which we call "time skewing", in Section 3. This article also includes a discussion of related work (Section 4) and concluding remarks (Section 5).

1.1 Definitions

For the discussion that follows, we will define the *machine balance* as the maximum sustainable rate of performing floating-point arithmetic (typically for data in registers) divided by the maximum sustainable data transfer rate for unit-stride accesses. The *machine balance* will therefore be a function of the location of the data in the memory hierarchy, and we can speak of the *L1 cache machine balance*, the *L2 cache machine balance*, the *main memory machine balance*. For distributed shared memory systems, there will be a corresponding *remote memory machine balance*, and for virtual memory machines there will be a *virtual memory machine balance*.

Corresponding to the *machine balance* parameters, we can define *compute balance* (or *loop balance*) parameters. The *compute balance* parameters are defined as the required number of floating-point operations divided by the re-

```
// 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 1: Matrix Multiplication

quired number of memory references to each level in the memory hierarchy. So we have *L1 cache compute balance*, *L2 cache compute balance*, *main memory compute balance*, etc. In some cases we may with to compare the number of operations to the total number of memory references—we call this ratio simply the compute balance.

In general, loop nests for which the *compute balance* parameter is greater than the *machine balance* parameter of the memory containing the data may be compute-bound; code sections for which the *compute balance* parameter is less than the *machine balance* parameter will be bandwidth-bound. When the compute balance is greater than the machine balance, optimizations that reduce latency (such as prefetching) may help to improve performance by moving some memory access delays to times where the processor is busy. However, latency reducing optimizations cannot compensate for inadequate memory bandwidth. The comparison of compute and machine balances can be repeated at each level of the memory hierarchy to examine how bottlenecks change with "distance" from the processor.

Compute balance can vary with the number of loops considered; for example, the `k` loop of the matrix multiply code in Figure 1 accesses accesses $3N$ memory locations and performs N multiplications and N additions, for a compute balance of $\frac{2}{3}$. Thus, we cannot only make use of CPU speed in excess of $\frac{2}{3}$ of the bandwidth of the memory in which these data reside at the start of the `k` loop, no matter what optimizations we apply to the `k` loop itself. The inner two loops have a compute balance of approximately $\frac{1}{2}$ (accessing N^2+2N memory locations during $2N^2$ operations). However, the entire loop nest has a compute balance of $\frac{N}{3}$ ($3N^2$ memory locations and N^3 operations). Thus, even machines with extremely high main memory balance (such as 100) may not be bandwidth bound during the multiplication of large arrays (such as 300x300).

Of course, simply matching compute balance to machine balance does not guarantee efficient processor use—other factors, such as cache interference, memory latency, or limits on cache size, may interfere. In such cases, it may be posible to counter these factors with compile-time optimizations: Copying can be used to reduce cache interference [LRW91, TGJ93], prefetching can reduce stalls due to latency [MLG92], and tiling [Wol89, WL91] can improve the performance of this loop nest when the three arrays will not fit entirely in cache.

However, a mismatch between compute balance and machine balance does identify cases in which bandwidth places limits on locality, no matter what we

```
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 2: Three Point "In-Place" Stencil (from [WL91, Wol92])

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

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

Figure 3: Three Point Stencil

do about interference or latency. If we a loop nest performs N^2 calculations on N^2 values, the only way to make this loop nest run faster than N^2 accesses to main memory is to ensure that the data are in cache before the nest starts–no amount of optimization of the loop nest itself can change this.

In this discussion, we have ignored the issue of whether or not two distinct references might refer to different addresses that share a same cache line. The calculations we consider traverse memory with unit stride; in such cases, the sharing of cache lines do not affect the question of whether or not the memory system has enough bandwidth to keep up with the CPU.

2 Stencils, Skewing, and Locality

The code shown in Figure 2, and a variant of this code with three loops, were used by Wolf and Lam to demonstrate their approach to optimizing data locality [WL91, Wol92]. Note that these codes have compute balances of $O(T)$, so for large values of T we may be able to achieve high degrees of locality without looking outside of this nest. Wolf and Lam show that high degrees of locality can be produced by first skewing the inner loop(s) to make the nest fully permutable, and then applying tiling.

The techniques of Wolf and Lam require perfectly nested loops, or loops that can be converted to a perfect nest using the techniques given in Section 2.7 of [Wol92]. The stencils shown in Figures 3 and 4 lie outside the domain of these techniques. However, if we align [ACK87] and fuse the two nests in the t loop, we can then skew and tile the loops and adjust the compute balance as we would for Figure 2.

This combination of unimodular and non-unimodular transformations can

```
for (int t = 0; t<T; t++)
  {
    for (int i = 0; i<ROWS; i++)
      for (int j = 0; j<COLS; j++)
        old[i][j] = cur[i][j];

    for (int i = 1; i<ROWS-1; i++)
      for (int j = 1; j<COLS-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 4: Five Point Stencil

be described consicely in the framework of [KP94]. In this framework, each iteration of each statement is identified with a unique tuple of integers. These integers may correspond to the loop index value of a surrounding loop, or they may indicate which of several statements is being performed. For example, if the loop nest in Figure 4 were the second statement in a function, we could identify iteration $t = 4, i = 7, j = 6$ of the assignment to `old` with the tuple $[2, 4, 1, 7, 1, 6, 1]$ (i.e. statement 2 of the function, $t = 4$, statement 1 in the `t` loop, $i = 7$, statement 1 in the `i` loop, $j = 6$, statement 1 in the `j` loop), or iteration $t = 5, i = 4, j = 8$ of the assignment to `cur` with $[2, 5, 2, 4, 1, 8, 1]$. 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. Using this system, we describe aforementioned transformation of Figure 4 as:

$$\{ [\, 2, t, 1, i, 1, j, 1 \,] \rightarrow [\, 2, t/B, 1, (2t + i - 1)/B, 1, j + 2t - 1, 1, (2t + i - 1)\%B, 1, t\%B, 1 \,] \}$$
$$\{ [\, 2, t, 2, i, 1, j, 1 \,] \rightarrow [\, 2, t/B, 1, (2t + i)/B, 1, j + 2t, 1, (2t + i)\%B, 1, t\%B, 2 \,] \}$$

where B is the blocksize, which must be known at compile time.

This transformation produces blocks that perform $O(COLS * B * B)$ operations while accessing $O(COLS * B)$ memory elements. Furthermore, only $O(B * B)$ values are live within the block at any time. Thus, in the absence of interference, we should achieve main-memory compute balance of $O(B)$, given a cache of size $O(B * B)$. We will return to the subject of interference in Section 4.

Note that it is not possible to achieve a main-memory compute balance above 3 for Figure 4 without exploiting locality between the two `i`/`j` nests (unless both array fit entirely in cache). Tiling the `i` and `j` loops that surround

the calculation may be necessary to produce this balance, if the cache cannot simultaneously hold two rows of both arrays.

3 Time Skewing

It is possible to reduce the number of main memory writes of the previous stencils by a further factor of two. Following the application of our algorithm to the code in Figure 4 is somewhat difficult. We therefore use the simpler one-dimensional stencil calculation shown in Figure 3 to illustrate our approach, and then give a general formulation.

We start by performing dataflow analysis of the `cur` and `old` arrays [Fea91, PW93, Won95]. The result of this analysis is a graph of the flow of values through the iteration space, which we call the iteration space dataflow graph. Each node represents a value produced by some iteration of some statement in the loop, and has incoming edges from the other nodes whose values that are used in the calculations. This graph is represented implicitly, and may be parameterized by the values that describe this space or the flow of values. Figure 5 shows the dataflow values for Figure 3, assuming `N=7` and `T=3` (ignore the shading for now). Note that we can perform this analysis without knowing `N` and `T`, but have stated them in this case to simplify the picture.

This representation can also be viewed as the result of array expansion (converting each definition of an array element `A[X]` at time `t` into a definition of `A[t][X]`, and adjusting array uses to match) and removal of temporary arrays (in this example, since `old` is not used after the loop, and `old[t][i]` is defined as `cur[t-1][i]`, we can replace the uses of `old` with uses of `cur`), as shown Figure 6. We use this notation in the following discussion (and in Figure 5), even though we do not actually store values in this way.

Note that only the values of `cur[T][*]` are needed when the loop is over. All other values can be placed temporarily in cache, and then overwritten without ever being stored in main memory, as long as we finish using each value before we overwrite it. In this example, we can traverse the dataflow graph from lower left to the upper right, keeping a band of values of thickness three in the cache (shown in light gray on Figure 5), and storing each final value in main memory as we generate it (dark gray). This traversal allows us to read one new value into cache (in this case, `cur[0][6]`) and combine it with other "intermediate" values already in the cache (in this case, `cur[0][5]`, `cur[0][4]`, `cur[1][3]`, and `cur[2][2]`) to produce a sequence of new intermediate and final values (`cur[1][5]`, `cur[2][4]`, and `cur[3][3]`), and then release some intermediate values that are no longer needed (`cur[0][3]`, `cur[1][2]`, and `cur[2][1]`).

This traversal lets us produce `T` new values while performing only one read from, and one write to, main memory (assuming we can keep the entire gray band in cache). If we cannot keep `3T` elements in cache at once, we block the `T` loop (again, let B be the block size).

This reordering of iterations is identical to that discussed in the previous section, except that we skew by `t` rather than `2t`. It produces code with the

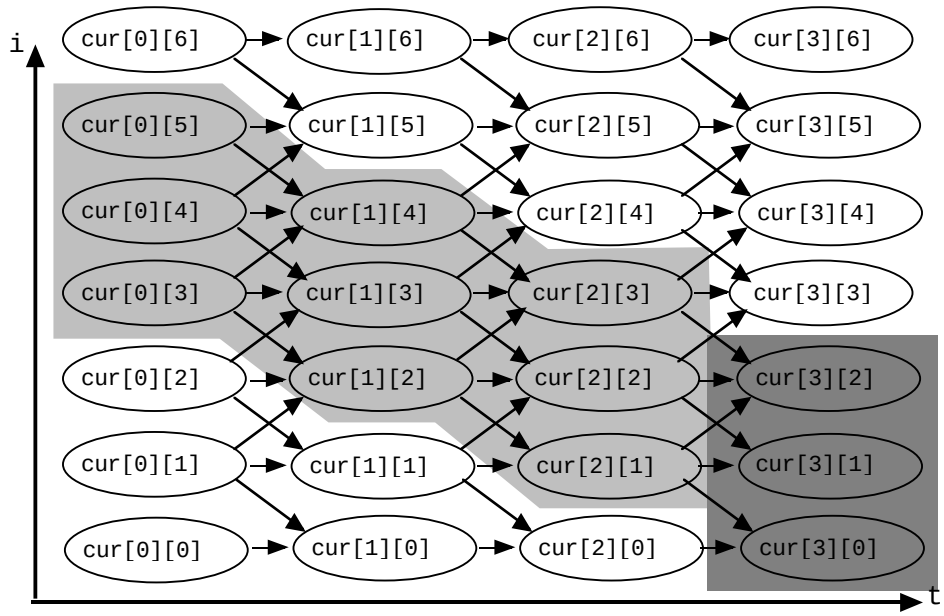

Figure 5: Dataflow Graph of Three Point Average Calculation

```
for (int t = 0; t<T; t++)
  {
    // compute new cur[t+1] values from the values in cur[t]
    cur[t+1][0] = cur[t][0];
    for (int i = 1; i<N-1; i++)
      {
        cur[t+1][i] = 1.0/3 *
          (cur[t][i-1] + cur[t][i] + cur[t][i+1]);
      }
    cur[t+1][N-1] = cur[t][N-1];
  }
```

Figure 6: Three Point Average Calculation in Single Assignment Form

same ratio of operations to values that are live across block boundaries. The only question that remains is where to store the values that are not live: Figure 5 suggests that every value is stored in a new location—obviously this will not improve memory locality. Instead, we create an array (named "cache") of size 3 by B to hold all values in the grey bar.

If all values in the grey bar are stored in the "cache" array, then each block reads N values from main memory, and writes N values into memory. The other $N(B-2)$ values exist only temporarily in the cache array. In contrast, if we transform the iterations without remapping the storage, we end up writing new values into both the `cur` and `old` arrays. The values stored in one of these arrays are dead, but all $2N$ values will still be written into main memory (in the absence of some mechanism to inform the cache hardware about dead values). Thus, our remapping of storage has reduced the number of writes to memory by a factor of two.

When transforming stencils of arrays of higher dimension, we must block all but the innermost loop (otherwise the tile size grows with N rather than B). When a stencil includes values that are more than one element away, we skew by larger factors of `t`. In general, for a nest with loops $i_1 ... i_n$ inside a time loop t, with dependence distances $\delta_1 ... \delta_n$, we perform the following transformation:

$$\{\ [t, i_1, i_2 ... i_n] \rightarrow [t/B, (i_1 + \delta_1 * t)/B, (i_2 + \delta_2 * t)/B, ..., (i_{n-1} + \delta_{n-1} * t)/B,$$
$$i_n + \delta_n * t,$$
$$(i_{n-1} + \delta_{n-1} * t)\%B, (i_2 + \delta_2 * t)\%B, ..., (i_1 + \delta_1 * t)\%B, t\%B]\ \}$$

(in the interest of simplicity, we have omitted the constant levels of the transformation, as they are unnecessary when only one statement is being transformed).

We can describe the mapping of values to memory locations with a similar notation: we describe a mapping from the statement iterations to the address expressions for each array. For our transformation, values produced at the end of a block of the time loop are stored in the original array (`cur`); values that are not live past the end of the block are stored in `cache`. We must also use

additional arrays to store the values produced at the ends of other blocked loops (called `tide` arrays here). We do $O(B)$ iterations before writing out a value to either the `cur` array or one of the `tide` arrays, so the introduction of these new arrays does not have a significant effect on the balance of the resulting computation. The total data mapping, in the the notation used in [SW98], is

$$
\begin{aligned}
&[tb, xb_1, ...xb_{n-1}, s_n, xx_1, ...xx_{n-1}, tt] \rightarrow \\
&\qquad \texttt{cur}[i_1, i_2...i_n], \text{when } tt = B-1. \\
&\qquad \texttt{tide}_j[s_n, xx_1, ...xx_{n-1}, tt], \text{when } xx_j + \delta_j + 1 \geq B \\
&\qquad\qquad\qquad\qquad \wedge (\nexists k > j \text{ s.t. } xx_k + \delta_k + 1 \geq B) \wedge tt \neq B-1, \\
&\qquad \texttt{cache}[s_n mod(\delta_n + 2), xx_1, ...xx_{n-1}, tt], \text{otherwise.}
\end{aligned}
$$

The algorithms given in [KPR95] and [SW98] can be used to generate code for the iteration spaces and array subscript expressions, given the above descriptions of the iteration space transformations and memory mappings. Note that [SW98] is the only data transformation framework that can represent this transformation: other frameworks apply transformations to all uses of a given array (rather than a single write statement), and require a single transformation (rather than a set of transformations, producing references to different arrays, that together cover all iterations of the statement being transformed).

4 Related Work

Current techniques for improving locality [GJ88, WL91, 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. That is, these techniques search for references that referring to the same array element, or to adjacent array elements (which may share a cache line), possibly in different iterations of a loop. These techniques then reorder calculations so that those references that share cache lines occur together. References to different arrays are often moved apart where possible, to reduce cache interference. Reordering generally is accomplished by combining loop distribution (to separate unrelated calculations) and loop tiling and interchange (to bring together accesses to the same cache line). [WL91] also apply loop skewing to enable tiling and interchange.

These techniques can be applied to a wide range of calculations, while we have only studied stencils. However, our technique can be applied to stencils that could not be optimized by previous techniques: they lie outside the domain of the algorithm of Wolf and Lam, and more recent work has overlooked loop skewing. Our full "time skewing" algorithm produces a smaller amount of main memory traffic than a mere reordering of the iterations without remapping of the arrays used, though at a cost of significant additional code complexity. This cost may not be worth the savings in memory traffic unless the processor is dramatically faster than memory. [SSW97] demonstrated that time skewing can be used to greatly increase the speed of calculations involving arrays that are too large to fit in main memory.

Other techniques for improving locality may be confounded by cache interference, especially when the the size of a row of the array is a multiple of the cache line size. This effect can be reduced by adjusting the array dimensions, or copying each tile into temporary storage before working with it [LRW91, TGJ93]. With our approach, cache interference will only arise from conflicts between our small "cache" array and the arrays we use to store values at the edges of blocks of iterations. Since the allocation of these arrays is controlled by our algorithm, we may be able to eliminate interference by reducing the size of our cache array, and allocating the other arrays so that the elements we use will fit in the unused cache lines. Even without such aggressive techniques, our algorithm should not cause much cache interference, since most of the work is done entirely within the "cache" array, which is by definition small enough to fit entirely within cache.

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 the two approaches cannot be used together.

5 Conclusion

We have demonstrated that the value of loop skewing in improving memory locality may only be evident in a compiler that includes transformations that have not been applied in studies of locality. Specifically, some stencils require that skewing be combined with either (a) the interchange of imperfectly nested loops, or (b) forward substitution of array expressions and a new form of data transformation. In other cases, it may be necessary to skew and block a nest with in which the outer loop is a `while` rather than a `for`, though we have not addressed this case here. As we noted at the end of Section 2, bandwidth places an upper limit on the memory locality that can be achieved without skewing with respect to the outer "time" loop. Thus, we conclude that loop skewing should not be overlooked in the search for transformations that improve memory locality.

We have also shown that data transformation, when used in combination with iteration space transformation, may be useful as a tool for further reducing memory traffic in stencil calculations. We formulated our approach by starting with a description of the data flow in the calculation (a fundamental characteristic of the algorithm), rather than by searching for locality based on the arrays used by the programmer (an artifact of the expression of the algorithm). While both approaches produce the same iteration space transformation, our approach does provide insight into what values actually need to be written out to main memory.

References

[ACK87] R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Conference Record of the Fourteenth ACM Symposium on Principles of Programming Languages*, pages 63–76, January 1987.

[Fea91] Paul Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–53, February 1991.

[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.

[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.

[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.

[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.

[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. 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).

[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] Michael Wolfe. More iteration space tiling. In *Proc. Supercomputing 89*, pages 655–664, November 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.