Cache-Efficient Multigrid Algorithms*

Sriram Sellappa[1] and Siddhartha Chatterjee[2]

[1] Nexsi Corporation, 1959 Concourse Drive, San Jose, CA 95131
Email: sriram.sellappa@nexsi.com
[2] Department of Computer Science, The University of North Carolina
Chapel Hill, NC 27599-3175
Email: sc@cs.unc.edu

Abstract. Multigrid is widely used as an efficient solver for sparse linear systems arising from the discretization of elliptic boundary value problems. Linear relaxation methods like Gauss-Seidel and Red-Black Gauss-Seidel form the principal computational component of multigrid, and thus affect its efficiency. In the context of multigrid, these iterative solvers are executed for a small number of iterations (2–8). We exploit this property of the algorithm to develop a cache-efficient multigrid, by focusing on improving the memory behavior of the linear relaxation methods. The efficiency in our cache-efficient linear relaxation algorithm comes from two sources: reducing the number of data cache and TLB misses, and reducing the number of memory references by keeping values register-resident. Experiments on five modern computing platforms show a performance improvement of 1.15–2.7 times over a standard implementation of Full Multigrid V-Cycle.

1 Introduction

The growing speed gap between processor and memory has led to the development of memory hierarchies and to the widespread use of caches in modern processors. However, caches by themselves are not a panacea. Their success at reducing the average memory access time observed by a program depends on statistical properties of its dynamic memory access sequence. These properties generally go under the name of "locality of reference" and can by no means be assumed to exist in all codes. Compiler optimizations such as *iteration space tiling* [13,12] attempt to improve the locality of the memory reference stream by altering the schedule of program operations while preserving the dependences in the original program. While the theory of such loop transformations is well-developed, the choice of parameters remains a difficult optimization problem.

The importance of locality of reference is even more critical for hierarchical computations based on techniques such as multigrid, fast multipole, and wavelets, which

* This work was performed when the first author was a graduate student at UNC Chapel Hill. This work is supported in part by DARPA Grant DABT63-98-1-0001, NSF Grants EIA-97-26370 and CDA-95-12356, The University of North Carolina at Chapel Hill, Duke University, and an equipment donation through Intel Corporation's Technology for Education 2000 Program. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

V.N. Alexandrov et al. (Eds.): ICCS 2001, LNCS 2073, pp. 107–116, 2001.

typically perform $\Theta(1)$ operations on each data element. This is markedly different from dense matrix computations, which perform $\mathcal{O}(n^\epsilon)$ operations per data element (with $\epsilon > 0$) and can profit from data copying [7]. The lack of "algorithmic slack" in hierarchical codes makes it important to reduce both *the number of memory references* and *the number of cache misses* when optimizing them. Such optimizations can indeed be expressed as the combination of a number of standard compiler optimizations, but even the best current optimizing compilers are unable to synthesize such long chains of optimizations automatically. In this paper, we apply these ideas to develop cache-efficient multigrid.

The remainder of the paper is organized as follows. Section 2 introduces the problem domain. Section 3 discusses cache-efficient algorithms for this problem. Section 4 presents experimental results. Section 5 discusses related work. Section 6 summarizes.

2 Background

Many engineering applications involve boundary value problems that require solving elliptic differential equations. The discretization of such boundary value problems results in structured but sparse linear systems $Av = f$, where v is the set of unknowns corresponding to the unknown variables in the differential equation and f is the set of discrete values of the known function in the differential equation. A is a sparse matrix, whose structure and values depend on the parameters of discretization and the coefficients in the differential equation. Since A has few distinct terms, it is generally represented implicitly as a stencil kernel.

Such systems are often solved using iterative solvers such as linear relaxation methods, which naturally exploit the sparsity in the system. Each iteration of a linear relaxation method involves refining the current approximation to the solution by updating each element based on the approximation values at its neighbors. Figure 1 shows three common relaxation schemes: Jacobi, Gauss-Seidel, and Red-Black Gauss-Seidel. We consider a two-dimensional five-point kernel that arises, for example, from the discretization of Poisson's equation on the unit square. Of these, the Jacobi method is generally not used as a component of multigrid because of its slow convergence and its additional memory requirements. We therefore do not consider it further.

The error in the approximate solution can be decomposed into oscillatory and smooth components. Linear relaxation methods can rapidly eliminate the oscillatory components, but not the smooth components. For this reason, they are generally not used by themselves to solve linear systems, butt are used as building blocks for multigrid [3]. Multigrid improves convergence by using a hierarchy of successively coarser grids. In the multigrid context, linear relaxation methods are called *smoothers* and are run for a small number of iterations (2–8). We call this quantity `NITER`.

In addition to the smoother, multigrid employs *projection* and *interpolation* routines for transferring quantities between fine and coarse grids. Figure 2 shows the Full Multigrid V-cycle algorithm that we consider in this paper. Of these three components, the smoother dominates in terms of the number of computations and memory references. (For `NITER` =4, we have found it to take about 80% of total time.)

(a) Five-point Jacobi

```
for (m = 0; m < NITER; m++) {
 for (i = 1; i < (N-1); i++)
  for (j = 1; j < (N-1); j++)
   U[i,j] = w1*V[i,j-1] + w2*V[i-1,j] + w3*V[i,j] +
            w4*V[i+1,j] + w5*V[i,j+1] + w6*f[i,j];
 Swap(U,V);
}
```

(b) Five-point Gauss-Seidel

```
for (m = 0; m < NITER; m++)
 for (i = 1; i < (N-1); i++)
  for (j = 1; j < (N-1); j++)
   V[i,j] = w1*V[i,j-1] + w2*V[i-1,j] + w3*V[i,j] +
            w4*V[i+1,j] + w5*V[i,j+1] + w6*f[i,j];
```

(c) Five-point Red-Black Gauss-Seidel

```
for (m = 0; m < NITER; m++) {
  offset = 1;
  for (i = 1; i < (N-1); i++) {
    offset = 1-offset;
    for (j = 1+offset; j < (N-1); j += 2) {
      V[i,j] = w1*V[i,j-1] + w2*V[i-1,j] + w3*V[i,j] +
               w4*V[i+1,j] + w5*V[i,j+1] + w6*f[i,j];
    }
  }

  offset = 0;
  for (i = 1; i < (N-1); i++) {
    offset = 1-offset;
    for (j = 1+offset; (j) < (N-1); j += 2) {
      V[i,j] = w1*V[i,j-1] + w2*V[i-1,j] + w3*V[i,j] +
               w4*V[i+1,j] + w5*V[i,j+1] + w6*f[i,j];
    }
  }
}
```

Fig. 1. Code for three common linear relaxation methods.

$MV^h(v^h, f^h)$

1. Relax ν_1 times on $A^h u^h = f^h$ with initial guess v^h.
2. If $\Omega^h \neq$ coarsest grid then
   $f^{2h} = Project(f^h - A^h v^h)$
   $v^{2h} = 0$
   $v^{2h} = MV^{2h}(v^{2h}, f^{2h})$
   $v^h = v^h + Interpolate(v^{2h})$.
3. Relax ν_2 times on $A^h u^h = f^h$ with initial guess v^h.
4. Return v^h.

(a) V-cycle Multigrid

{Initialize v^h,v^{2h},... to zero}

$FMV^h(v^h, f^h)$

1. If $\Omega^h \neq$ coarsest grid then
   $f^{2h} = Project(f^h - A^h v^h)$
   $v^{2h} = 0$
   $v^{2h} = FMV^{2h}(v^{2h}, f^{2h})$
   $v^h = v^h + Interpolate(v^{2h})$.
2. $v^h = MV^h(v^h, f^h)$ ν_0 times.
   /* Invoke V-cycle ν_0 times to refine the solution */
3. Return v^h.

(b) Full Multigrid V-cycle

Fig. 2. Multigrid algorithms. Ω^h is a grid with grid spacing h. A superscript h on a quantity indicates that it is defined on Ω^h.

We now consider the memory system behavior of smoothers in terms of the 3C model [6] of cache misses.

- The classical Gauss-Seidel algorithm makes `NITER` sweeps over the whole array (2*`NITER` sweeps in the case of Red-Black Gauss-Seidel), accessing each element `NITER` times. Accesses to any individual element are temporally distant; since the array size is larger than the capacity of the cache, the element is likely to have been evicted from the cache before its access in the next iteration. The multiple sweeps of the array thus result in *capacity misses* in the data cache.
- The computation at an element in the array involves the values at the adjacent elements. So there is some spatial locality in the data. But the data dependences make it difficult for compilers to exploit this spatial locality.
- There could be *conflict misses* between the `V` and `f` arrays in Figure 1.
- The repetitive sweeps across the array cause address translation information to cycle through the (highly associative) TLB, which is deleterious to its performance. As the matrix dimension n grows, a virtual memory page will hold only $\Theta(1)$ rows or columns, requiring $\Theta(n)$ TLB entries to map the entire array. The resulting *capacity misses* in the TLB can be quite expensive given the high miss penalty.

The above observations motivate the algorithmic changes described in Section 3 that lead to cache-efficient multigrid algorithms.

3 Cache-Efficient Multigrid Algorithms

Our improvements to the efficiency of FMV stem exclusively from improvements to the memory behavior of the underlying smoothers. Two characteristics of these schemes are critical in developing their *cache-efficient* versions. First, we exploit the fact that the relaxation is run for a small number of iterations (2–8) by employing a form of iteration-space tiling [13] to eliminate the capacity misses incurred by the standard algorithm. Second, we exploit the spatial locality in the relaxation by retaining as many values in the registers as possible, using *stencil optimization* [4] to reduce the number of memory references. We describe our cache-efficient algorithms for two-dimensional, 5-pt Gauss-Seidel and Red-Black Gauss-Seidel schemes. We call these cache-efficient algorithms *temporal blocking* algorithms [2], because they partition the array into blocks and process blocks lexicographically to enhance temporal proximity among memory references. Note that these techniques preserve all data dependences of the standard (cache-unaware) algorithm. **Hence our cache-efficient algorithm is numerically identical to the standard algorithm.**

3.1 Cache-Efficient Gauss-Seidel Algorithm

The key idea in *temporal blocking* is to smoothen a subgrid of the solution matrix `NITER` times before moving on to the next subgrid; this clusters the `NITER` accesses to a particular element in time. We choose the subgrid size to fit in L1 cache; hence there are no capacity misses, as long as we touch only the elements within that subgrid, while working on that subgrid. Subgrids are square, of size $K * K$; boundary subgrids

are possibly rectangular. Gauss-Seidel requires elements to be updated in lexicographic order, requiring subgrids to also be visited the same way.

Consider the lowermost leftmost subgrid. All the elements of the subgrid can be updated once, except the elements at the right and top boundaries (to update them we need their neighbors, some of which lie outside the subgrid). Similarly, among the elements that were updated once, all the elements—except those on the right and top boundaries—can be updated again. Thus, for each additional iteration, the boundary of the elements with updated values shrinks by one along both dimensions. As a result, we have a *wavefront* of elements of width NITER that were updated from 1 to NITER-1 times. This wavefront propagates from the leftmost subgrid to the rightmost subgrid and is absorbed at the boundary of the matrix, through overlap between adjacent subgrids. Figure 3(b) shows the layout of overlapping subgrids, with NITER +1 rows and columns of overlap. The effect of NITER relaxation steps is illustrated for a subgrid in Figure 3(a) and for the entire matrix in Figure 4.

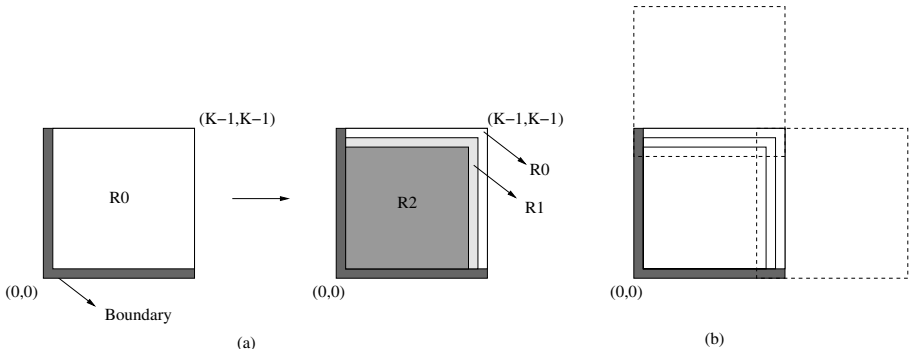


Fig. 3. (a) Transformation of the lowermost-leftmost subgrid by the *temporal blocking* algorithm for NITER =2. R0 is the set of elements that have not been updated, R1 is the set of elements that have been updated once, and R2 is the set of elements that have been updated twice. (b) The layout of the overlapping subgrids in the matrix.

The temporal blocked algorithm and the standard algorithm are numerically identical. The important performance difference between them comes from their usage of the memory system. Each subgrid is brought into the L1 cache once, so working within a subgrid does not result in capacity misses. There is some overlap among subgrids, and the overlapping regions along one dimension are fetched twice. Since NITER is 2–8, the overlapping region is small compared to the subgrid size, and the temporal blocking algorithm effectively makes a single pass over the array, *independent of NITER* . In contrast, the standard algorithm makes NITER passes over the array even if a compiler tiles the two innermost loops of Figure 1(b).

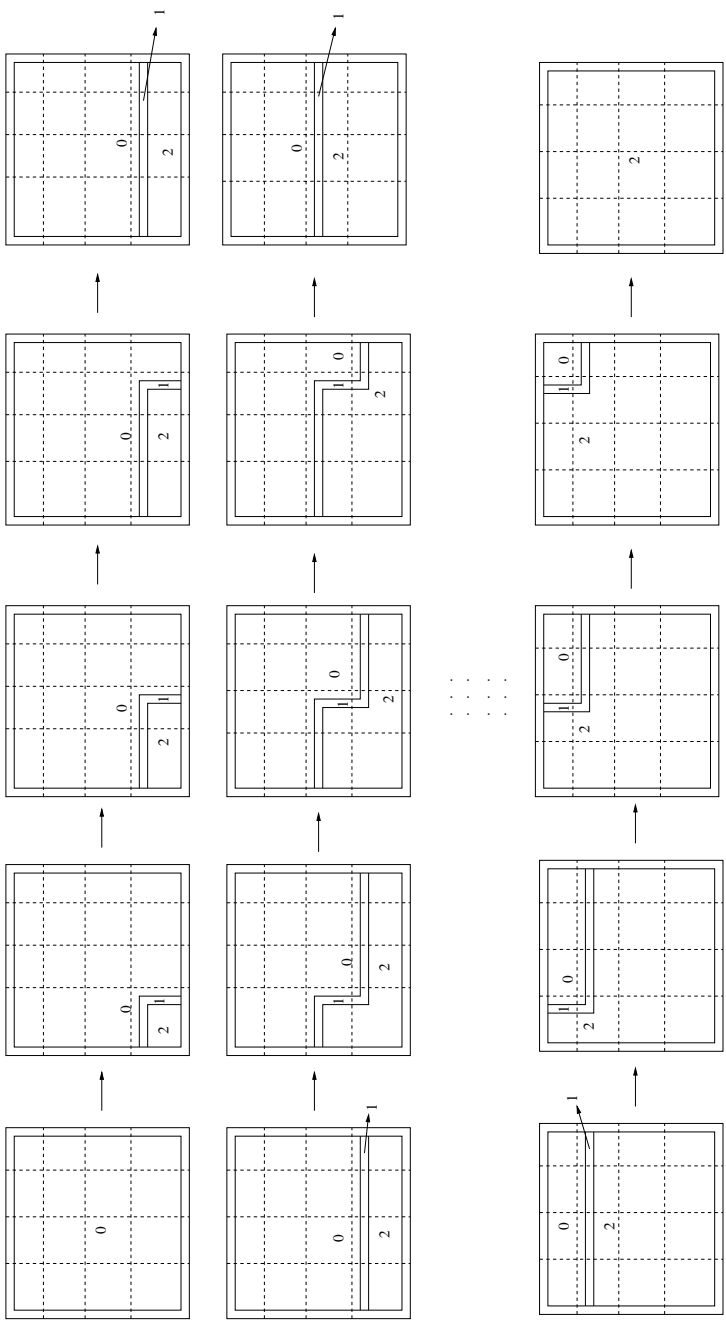


Fig. 4. Operation of the temporal blocking algorithm for Gauss-Seidel for `NITER` = 2. The initial matrix is the lowermost-leftmost matrix, and the final matrix is the rightmost-topmost matrix.

3.2 Stencil Optimization

Temporal blocking propagates the wavefront in a subgrid and pushes it to the beginning of the next subgrid. This shifting of the wavefront by one column at a time is a stencil operation where each element is updated using its neighbors and the elements are updated in lexicographic order. Each element of the subgrid is referenced five times in a single iteration of the `m`-loop in Figure 1(b): once for updating each of its four neighbors and once for updating itself. Note that, except for debugging situations, the intermediate values of the `V` array are not of interest; we care only about the final values of the elements after performing `NITER` steps of relaxation. This suggests that we might be able to read in each element value once, have it participate in multiple updates (to itself and to its neighbors) while remaining register-resident, and write out only the final updated value at the end of this process. If the value of `NITER` is small and the machine has enough floating-point registers, then this optimization is in fact feasible. What we have to do is to explicitly manage the registers as a small cache of intermediate results.

Performing stencil optimization at the source level requires care in programming (using explicit data transfers among several scalar variables) and availability of registers. Given the small value of `NITER`, the live variables fit within the register files available on most modern machines, and hence stencil optimization is very effective.

3.3 Cache-Efficient Red-Black Gauss-Seidel

Temporal blocking for Red-Black Gauss-Seidel is similar to that for Gauss-Seidel. The only difference is that the edges of the wavefront in this algorithm are sawtooth lines rather than straight lines, for the following reason. As we need the updated red elements to update the black elements, the boundary of the maximum number of elements that can be updated once is determined by the red elements in the subgrid, and the line joining the red elements has a sawtooth pattern. As a result, the width of the wavefront is 2*`NITER`. Other details of temporal blocking, like the propagation of the wavefront, remain unchanged. Stencil optimizations discussed above also apply in this case.

4 Experimental Results

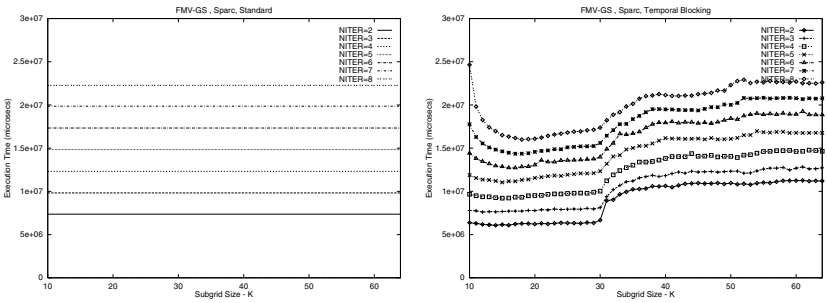
In this section we compare the performance of the standard and cache-efficient implementations of Full Multigrid V-cycle (FMV) with experimental results on a number of machines. We experimented on five commonly used modern computing platforms—UltraSPARC 60, SGI Origin 2000, AlphaPC 164LX, AlphaServer DS10, and Dell workstation—with both Gauss-Seidel and Red-Black Gauss-Seidel smoothers. Our test case is a two-dimensional Poisson's problem of size 1025×1025, with $\nu_0 = 4$ and $\nu_1 = \nu_2$ = `NITER` in Figure 2. The temporal blocking algorithm has one other parameter: K, the height of the subgrid. We are primarily interested in execution times of the algorithms. We use L1 cache misses, L2 cache misses, and TLB misses to explain the trends in execution time. Table 1 summarizes the overall performance improvement across platforms. For lack of space, we analyze the experimental data only for FMV with Gauss-Seidel relaxation on the Sparc.

Table 1. Ratio of running time of the standard version of FMV to the running time of the cache-efficient version, for Gauss-Seidel and Red-Black Gauss-Seidel relaxation schemes, on five modern computing platforms. The test problem is a two-dimensional Poisson problem of size 1025×1025. Larger numbers are better.

Platform	CPU	Clock speed	Gauss-Seidel	Red-Black Gauss-Seidel
UltraSPARC 60	UltraSPARC-II	300 MHz	1.35	2.4
SGI Origin 2000	MIPS R12000	300 MHz	1.35	2.4
AlphaPC 164LX	Alpha 21164	599 MHz	2.2	2.7
AlphaServer DS10	Alpha 21264	466 MHz	2.2	2
Dell Workstation	Pentium II	400 MHz	1.15	2

Figures 5(a) and (b) plot subgrid size *vs.* running time on the Sparc, one curve for each value of `NITER`. The plots demonstrate that the temporal blocking algorithm runs about 35% faster than the standard algorithm.

The plots in Figure 5(b) show an increase in running time of the cache-efficient FMV as the subgrid size increases, which is explained by TLB misses. All memory hierarchy simulations were performed using Lebeck's *fast-cache* and *cprof* simulators [8], for `NITER` = 4. Figure 6(a) shows the plot of TLB misses, which correlates with the degradation in running times for large subgrid sizes. The reason for the increase in the TLB misses is as follows. Since the size of the solution array is large, each row gets mapped to one or more virtual memory pages. When the temporal blocking algorithm works within a subgrid, the TLB needs to hold all the mapping entries of elements in that subgrid in the solution array (and the array of function values) in order to avoid additional TLB misses. Beyond a particular grid size, the number of TLB entries required exceeds the capacity of the TLB.



(a) (b)

Fig. 5. FMV with Gauss-Seidel relaxation, N=1025, and ν_0=4 on the Sparc. (a) Running time, standard version. (b) Running time, temporal blocked version.

Figure 6(b) shows the L1 cache misses on the Sparc. While the temporal blocking algorithm has fewer cache misses than the standard algorithm, the number of L1 cache misses increases with increase in subgrid size. Figures 6(c) and (d) show that conflict

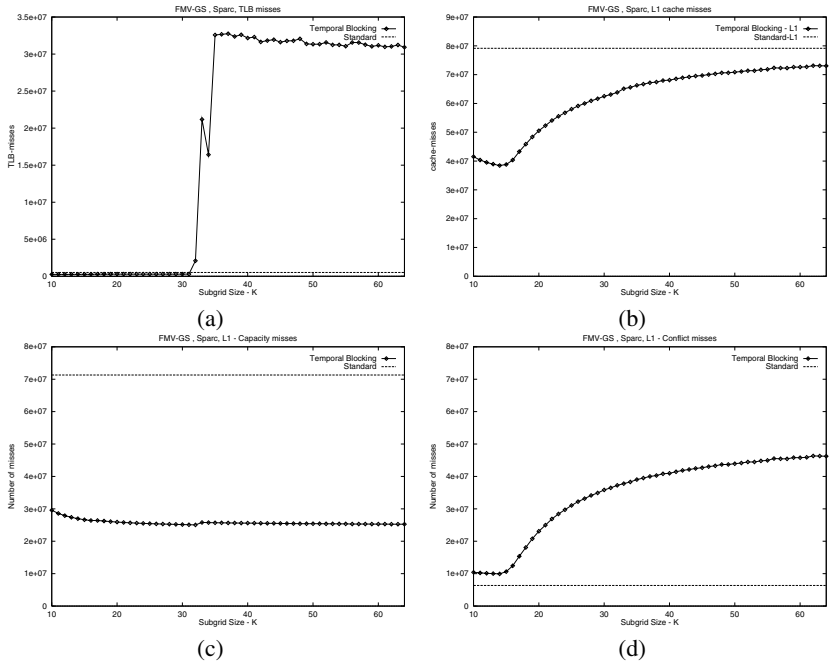


Fig. 6. FMV with Gauss-Seidel relaxation, N=1025, and ν_0=4 on the Sparc. (a) Number of TLB misses, `NITER` = 4. (b) Number of L1 cache misses, `NITER` = 4. (c) Number of L1 capacity misses, `NITER` = 4. (d) Number of L1 conflict misses, `NITER` = 4.

misses cause this increase. We confirmed that the conflict misses are due to *cross interference* between the V and f arrays, by running a cache simulation for a version of the code without the reference to f in the stencil. L1 cache misses remained constant in this simulation.

5 Related Work

Leiserson *et al.* [9] provide a graph-theoretic foundation for efficient linear relaxation algorithms using the idea of *blocking covers*. Their work, set in the context of out-of-core algorithms, attempts to reduce the number of I/O operations. Bassetti *et al.* [2] investigate stencil optimization techniques in a parallel object-oriented framework and introduce the notion of temporal blocking. In subsequent work [1], they integrate the blocking covers [9] work with their framework for the Jacobi scheme. Stals and Rüde [11] studied program transformations for the Red-Black Gauss-Seidel method. They explore blocking along one dimension for two-dimensional problems, but our work involves two-dimensional blocking. Douglas *et al.* [5] investigate cache optimizations for structured and unstructured multigrid. They focus only on the Red-Black Gauss-Seidel relaxation scheme, Povitsky [10] discusses a different wavefront approach to a cache-friendly algorithm to solve PDEs.

Bromley *et al.* [4] developed a compiler module to optimize stencil computations on the Connection Machine CM-2. To facilitate this, they worked with a particular style of specifying stencils in CM Fortran. They report performance of over 14 gigaflops. Their work focuses on optimizing a single application of a stencil, but does not handle the repeated application of a stencil that is characteristic of multigrid smoothers. Moreover, their technique does not handle cases when the stencil operations are performed in a non-simple order, like the order of updates in Red-Black Gauss-Seidel.

6 Conclusions

We have demonstrated improved running times for multigrid using a combination of algorithmic ideas, program transformations, and architectural capabilities. We have related these performance gains to improved memory system behavior of the new programs.

References

1. F. Bassetti, K. Davis, and M. Marathe. Improving cache utilization of linear relaxation methods: Theory and practice. In *Proceedings of ISCOPE'99*, Dec. 1999.
2. F. Bassetti, K. Davis, and D. Quinlan. Optimizing transformations of stencil operations for parallel object-oriented scientific frameworks on cache-based architectures. In *Proceedings of ISCOPE'98*, Dec. 1998.
3. W. L. Briggs. *A Multigrid Tutorial*. SIAM, 1987.
4. M. Bromley, S. Heller, T. McNerney, and G. L. Steele Jr. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 145–156, Toronto, Canada, June 1991.
5. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss. Cache optimization for structured and unstructured grid multigrid. *Electronic Transactions on Numerical Analysis*, 10:21–40, 2000. University of Kentucky, Louisville, KY, USA. ISSN 1068–9613.
6. M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Trans. Comput.*, C-38(12):1612–1630, Dec. 1989.
7. M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Apr. 1991.
8. A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, 27(10):15–26, Oct. 1994.
9. C. E. Leiserson, S. Rao, and S. Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. *J. Comput. Syst. Sci.*, 54(2):332–344, 1997.
10. A. Povitsky. Wavefront cache-friendly algorithm for compact numerical schemes. Technical Report 99-40, ICASE, Hampton, VA, Oct. 1999.
11. L. Stals and U. Rüde. Techniques for improving the data locality of iterative methods. Technical Report MRR 038-97, Institut für Mathematik, Universität Augsburg, Augsburg, Germany, Oct. 1997.
12. M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Canada, June 1991.
13. M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing'89*, pages 655–664, Reno, NV, Nov. 1989.