Memory Characteristics of Iterative Methods*

Christian Weiß†, Wolfgang Karl†, Markus Kowarschik††, Ulrich Rüde††

† Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Technische Universität München, Germany
{weissc,karlw}@in.tum.de

†† Lehrstuhl für Systemsimulation (IMMD X)
Universität Erlangen-Nürnberg, Germany
{kowarschik,ruede}@informatik.uni-erlangen.de

Abstract

Conventional implementations of iterative numerical algorithms, especially multigrid methods, merely reach a disappointing small percentage of the theoretically available CPU performance when applied to representative large problems. One of the most important reasons for this phenomenon is that the current DRAM technology cannot provide the data fast enough to keep the CPU busy. Although the fundamentals of cache optimizations are quite simple, current compilers cannot optimize even elementary iterative schemes. In this paper, we analyze the memory and cache behavior of iterative methods with extensive profiling and describe program transformation techniques to improve the cache performance of two- and three-dimensional multigrid algorithms.

1 Introduction

Multigrid methods [11, 5] are among the most attractive algorithms for the solution of large sparse systems of equations that arise in the solution of elliptic partial differential equations (PDEs). However, even simple multigrid codes on structured grids with constant coefficients cannot exploit a reasonable fraction of the floating point peak performance of current microprocessors. Furthermore, the performance typically drops dramatically with growing problem size. Figure 1, for example, shows the floating point performance of a simple and straightforward FORTRAN77 multigrid code on several modern workstations[1]. The code implements a two-dimensional multigrid V-cycle on a structured grid with linear interpolation, half injection as the restriction operator and a red-black Gauss-Seidel smoother (four pre- and no postsmoothing iterations) using a 5-point finite difference discretization of the differential operator. Although the code is designed to make it as easy as possible for a compiler to apply optimizations, the performance for large grids is nevertheless disappointing on all platforms.

The reason for the disappointing performance is that current DRAM technology cannot transfer data between processor and main memory as fast as necessary to avoid idle periods of the processor. Hence, the CPU is stalled most of the time waiting for data coming from main memory. In the past the processor performance has approximately doubled every 18 months, whereas memory performance has doubled merely every seven years. Recently, the focus of computer designers has shifted towards increasing the memory bandwidth. For example, the Alpha 21264-based Compaq XP1000 machine [10] has an increased peak memory bandwidth of 2 GByte/sec compared to its predecessor Digital PWS, which uses an Alpha 21164 chip and has a main memory bandwidth of roughly 900 MByte/sec (see Figure 3). The sustainable bandwidth of user programs, however, is still far away from that. For example, the STREAM bandwidth [16] for the Compaq XP1000 (500 MHz) is only 745 MByte/sec and the achievable bandwidth for other architectures might be even much lower (see Figure 2). However, memory bandwidth is not the only cause for the performance gap between processor and main memory. The second source of this gap is the latency of the main memory DRAM modules.

*This project is partially funded by DFG Ru 422/7-1,2.

[1] All benchmarks in the article were compiled with native FORTRAN77 compilers and aggressive optimizations enabled. On the Intel platform we used *egcs* (V2.91.60). The platforms include an Intel PentiumII Xeon PC (450 MHz, 450 MFLOPS), a SUN Ultra 60 (296 MHz, 592 MFLOPS), a HP SPP2200 Convex Exemplar Node (200 MHz, 800 MFLOPS), a Compaq PWS 500au (500 MHz, 1 GFLOPS), and a Compaq XP1000 (500 MHz, 1 GFLOPS).

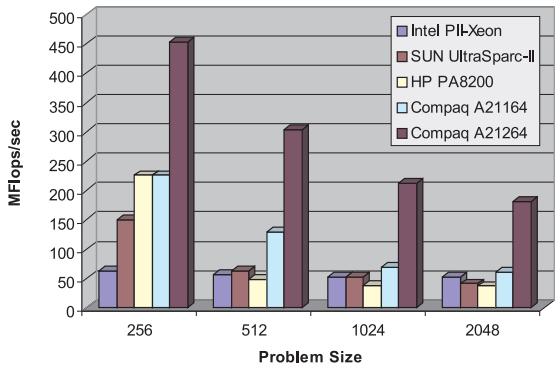


Figure 1: MFLOPS of a straightforward multigrid code in 2D.

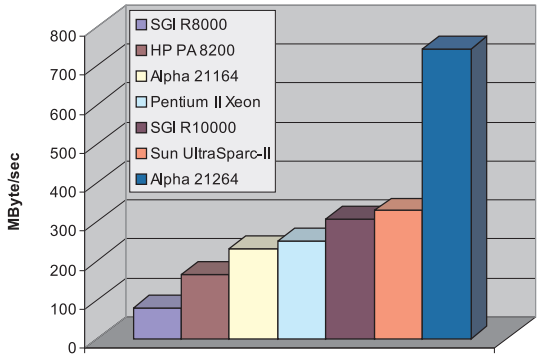
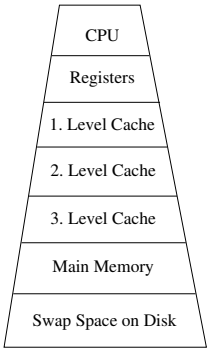


Figure 2: STREAM bandwidth of several computers in MByte/sec.



Capacity	Bandwidth	Latency
256 Bytes	24000 MB/s	2 ns
8 KBytes	16000 MB/s	2 ns
96 KBytes	8000 MB/s	6 ns
2 MBytes	888 MB/s	24 ns
1536 MBytes	1000 MB/s	112 ns

Figure 3: Memory hierarchy of the DEC Alpha PWS 500au based on the A21164 chip.

A common approach in modern computers to hide that performance gap is to use a hierarchy of memories consisting of comparatively fast but small caches at the top of the hierarchy and the slow but large main memory at the bottom of the hierarchy [15]. For example the memory hierarchy of the Digital PWS 500au shown in Figure 3 consists of three levels of caches. In order to reduce the data access latency, the L1 and L2 cache memory are located on-chip. However, since caches are designed to exploit spatial and temporal locality of memory references [12], they can only speed up accesses to frequently and recently used data. Iterative methods repeatedly perform global sweeps through data structures which are typically too large to fit completely in one of the caches during the computation. The consequence of repeatedly performing global sweeps is a high number of capacity misses, which dramatically reduces the efficiency of caches to speed up data accesses.

Cache optimization techniques — including data access transformations [3] like *loop interchange, loop fusion, loop blocking,* and *prefetching* as well as data layout transformations [17] like *array padding* and *array merging* — have proven their efficiency to optimize cache hit rates for example for matrix multiplication algorithms [4], linear algebra packages for dense matrices [1], and FFT algorithms [9]. Compilers are already able to automatically generate code for those problems that run nearly with peak performance on cache based architectures. However, current compiler technology is apparently not able to successfully apply those techniques to iterative methods. Some research in the area of cache optimization for iterative methods has been done by U. Rüde et al. [19], C.C. Douglas et al. [6], D.E. Keyes et al. [14], and D. Quinlan et al. [8]. Research to automatically perform data locality optimizations has been done in the SUIF project [21] and by Banerjee et al. [13], but it seems that the techniques are still far away from being applied to more complex methods, like multigrid, for example.

The article is organized as follows. The next section presents a detailed analysis of the cache behavior of the red-black Gauss-Seidel relaxation algorithm which is the most time-consuming part of our multigrid method. A detailed profiling measurement of the memory hierarchy of the Digital PWS500au explains why iterative methods perform poorly on cache based architectures. In Section 3 we demonstrate how data access transformations and data layout transformations can be applied to improve the cache behavior of the red-black Gauss-Seidel method. In Section 4 we show how those techniques can be applied to three-dimensional multigrid codes. Finally, we conclude with some remarks on future work.

2 Cache Behavior of Red-Black Gauss-Seidel

In the beginning, we consider a standard implementation of a two-dimensional red-black Gauss-Seidel relaxation method based on a 5-point discretization of the Laplace operator, as shown in Figure 4. The runtime behavior of our standard red-black Gauss-Seidel program on a Digital PWS 500au is summarized in Table 1. For the smallest grid size the floating point performance is relatively high compared to the peak performance of 1 GFLOPS. With growing grid size the performance increases slightly to more than 450 MFLOPS. Reaching a grid size of 128×128, however, the performance dramatically drops to 200 MFLOPS. For even larger grids ($>512 \times 512$) the performance further deteriorates below 60 MFLOPS.

To detect why those performance drops occur, we profiled the program using the *Digital Continuous Profiling Infrastructure* (*DCPI* [2]). The result of the analysis is a breakdown of CPU cycles spent for execution (Exec), nops, and different kinds of stalls (see Table 1). Possible causes of stalls are data cache misses (Cache), data table lookaside buffer misses (DTB), branch mispredictions (Branch), and register dependencies (Depend). For the smaller grid sizes the limiting factors are branch misprediction and register dependencies. However, with growing grid size, the cache behavior of the algorithm seems to have an enormous impact on the runtime. Thus, for the largest grids data cache miss stalls account for more than 80 % of all CPU cycles.

Since data cache misses are the dominating factor for the disappointing performance of our standard red-black Gauss-Seidel code, it seems reasonable to take a closer look at its cache behavior. Table 2 shows how many per cent of all array references are satisfied by the corresponding levels of the memory hierarchy. To obtain the data we counted the total number of array references which occur in the relaxation method and measured the number of L1 data cache accesses as well as the number of cache misses for each level of the memory hierarchy using *DCPI*. The difference between the measured and the estimated number of L1 data cache accesses is shown in column "$\pm$". Small values can be interpreted as measurement errors. Higher values, however, indicate that some of the array references are not implemented as loads or stores, but as very fast register accesses. The number of references which are satisfied by a particular level of the memory hierarchy is the difference between the number of accesses into it (misses of the memory level above it) and the number of accesses which are not satisfied by it (misses for that particular memory level). For example, the number of references satisfied by the L2 data cache is the number of L1 data cache misses minus the number of L2 data cache misses.

The analysis clearly shows that for the 32×32 and 64×64 grids the algorithm can access all of the data from the L1 or the L2 cache. However, as soon as the data does no longer fit in the L2 cache a high fraction of the data has to be fetched

```
double u(0:n,0:n), f(0:n,0:n)

do it = 1, noIter
   // red nodes:
   do i = 1, n-1
    do j = 1+(i+1)%2, n-1, 2
      Relax( u(i,j) )
    enddo
   enddo
   // black nodes:
   do i = 1, n-1
    do j = 1+i%2, n-1 , 2
      Relax( u(i,j) )
    enddo
   enddo
enddo
```

Figure 4: Standard implementation of red-black Gauss-Seidel.

Grid Size	MFLOPS	% of cycles used for					
		Exec	Cache	DTB	Branch	Depend	Nops
16	347.0	60.7	0.3	2.6	6.7	21.1	4.5
32	354.8	59.1	10.9	7.0	4.6	11.0	5.4
64	453.9	78.8	1.4	15.7	0.1	0.0	4.2
128	205.5	43.8	6.3	47.5	0.0	0.0	2.4
256	182.9	31.9	60.6	4.2	0.0	0.0	3.3
512	63.7	11.3	85.2	2.2	0.0	0.0	1.2
1024	58.8	10.5	85.9	2.4	0.0	0.0	1.1
2048	55.9	10.1	86.5	2.4	0.0	0.0	1.1

Table 1: Runtime behavior of red-black Gauss-Seidel.

Grid Size	Data Set Size	% of all accesses which are satisfied by				
		$\pm$	L1 Cache	L2 Cache	L3 Cache	Memory
32	17 KByte	4.5	63.6	32.0	0.0	0.0
64	66 KByte	0.5	75.7	23.6	0.2	0.0
128	260 KByte	-0.2	76.1	9.3	14.8	0.0
256	1 MByte	5.3	55.1	25.0	14.5	0.0
512	4 MByte	4.9	29.9	50.7	7.3	7.2
1024	16 MByte	5.1	27.8	50.0	9.9	7.2
2048	64 MByte	4.5	30.3	45.0	13.0	7.2

Table 2: Memory access behavior of red-black Gauss-Seidel.

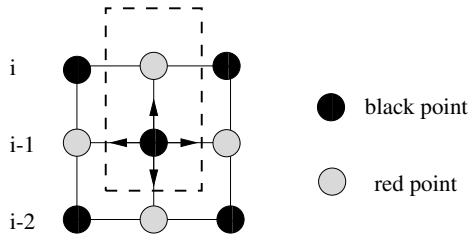


Figure 5: Data dependencies in a red-black Gauss-Seidel algorithm.

from the L3 cache. Similarly, for grids larger than 256×256, the data does not fit completely in the L3 cache. Obviously, the memory hierarchy cannot keep all of the data close to the CPU when the size of data grows.

The standard red-black Gauss-Seidel algorithm performs repeatedly one complete sweep through the grid from bottom to top updating all the red nodes and then another complete sweep updating all the black nodes. Assuming that the grid is too large to fit in the cache, the data of the lower part of the grid is no longer in the cache after the red update sweep because it has been replaced by the grid points belonging to the upper part of the grid. Hence, the data must be reloaded from the slower main memory into the cache again. In this process newly accessed nodes replace the upper part of the grid points in the cache, and as a consequence they have to be loaded from main memory once more.

Although the red-black Gauss Seidel method performs global sweeps through the data set, caches can nevertheless exploit at least some temporal and spatial locality [21]. For example, if we update the black node shown in the middle of Figure 5, we need the data of all the adjacent red nodes (which appear gray in Figure 5), the black node itself and the corresponding value of the right hand side of the equation (RHS). The values of the red points in lines *i-1* and *i-2* should be in the cache because of the update of the black points in row *i-2*. However, this is only true if at least two grid rows fit in the cache simultaneously. Also, the updated black node in line *i-1* might be in the cache if the black node and the red node on its left side belong to the same cache line [12]. The same argument holds for the red node in line *i* and the RHS value. Hence, the red node in line *i*, the RHS, and the black node in line *i-1* have to be loaded from memory whenever a cache line border is crossed, which means that the data has not yet been fetched into the cache before.

Table 2 motivates two goals of data locality optimizations for iterative methods. First, we must reduce the number of values which are loaded from the lowest levels of the memory hierarchy. In the case of a 512×512 grid, the grid data is held in main memory. The second goal is to fetch a higher fraction of the data out of one of the higher levels of the memory hierarchy, especially the registers and the L1 cache.

So far, we have only demonstrated the data access behavior for iterative methods with the red-black Gauss Seidel algorithm. However, other iterative methods have similar data locality properties. Iterative methods typically perform successively global sweeps through their data structures and have therefore a high potential of data reuse. Furthermore, iterative algorithms often reuse data during an update of a node which was accessed while updating a neighboring node. However, since the data structures are too large to fit in any cache, data locality is typically not exploited as well as possible. The result of this is poor cache behavior and consequently a poor overall performance.

3 Cache Optimization for Red-Black Gauss-Seidel in 2D

The key idea behind data locality optimizations is to reorder the data accesses so that as few of them as possible are performed between any two data references to the same memory location. With this, it is more likely that the data is not evicted from the cache and thus can be loaded from one of the higher levels of the hierarchy. However, the new access order is only valid if data dependencies are observed. In this paper, we thus focus on program transformations which maintain bitwise compatibility. The numerical results of our improved programs are identical to those obtained by the original algorithms.

3.1 Fusion and Blocking Technique

The data dependencies of the red-black Gauss-Seidel algorithm depend on the type of discretization which is applied. If a 5-point stencil is placed over one of the black nodes as shown in Figure 5, then all of the red points that are required for relaxation are up to date provided the red node above it is up to date. Consequently, we can update the red nodes in a row *i*

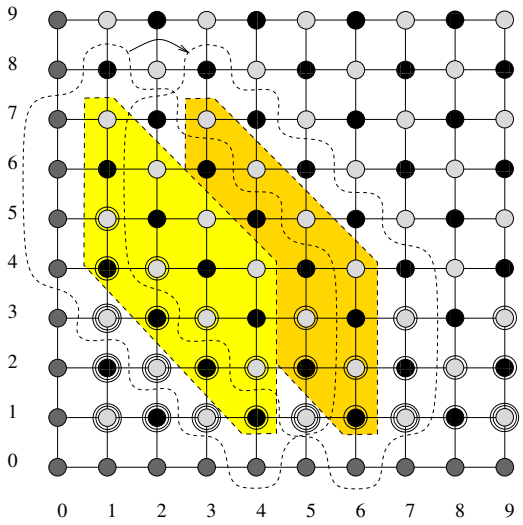


Figure 6: Two-dimensional blocking technique for red-black Gauss-Seidel.

and the black nodes in row *i-1* in pairs. This technique is called *fusion technique*. It fuses two consecutive sweeps through the grid, which update the red and black points separately, together to one sweep through the grid. However, some special treatments must be used for the update of the red nodes in the first and the update of the black nodes in the last row of the grid.

This technique applies only to one single red-black Gauss-Seidel sweep. If several successive red-black Gauss-Seidel iterations must be performed, the data in the cache is not yet reused from one sweep to the next, if the grid is too large to fit entirely in the cache. If a 5-point stencil is placed over one of the red nodes in line *i-2*, the node can be updated for the second time provided that all neighboring black nodes have been updated once. This is the case, as soon as the black node in line *i-1* directly above the red node has been touched once. As described before, this black node in turn can be updated as soon as the red node in line *i* directly above it has been updated for the first time. Consequently, we can update the red nodes in rows *i* and *i-2* and the black nodes in rows *i-1* and *i-3* in pairs. This technique — called *blocking technique* — can be generalized to more than just two successive red-black Gauss-Seidel sweeps.

Both of the above techniques require that a certain number of rows fit entirely in the cache. The *fusion technique* assumes that the cache can hold at least four rows of the grid. The *blocking technique* assumes that at least *m*2+2* rows of the grid fit in the cache, if *m* successive sweeps through the grid are performed together. Hence, they can reduce the number of accesses to slow memory, but fail to utilize the higher levels of the memory hierarchy efficiently, in particular the registers and the L1 cache. A high utilization of the registers and the L1 cache, however, is crucial for the performance of iterative methods. In the following, we therefore propose a two-dimensional blocking strategy. This requires special consideration, because the data dependencies of the red-black Gauss-Seidel method are not as simple as in a matrix multiplication (see [4] for an example).

3.2 Two-Dimensional Blocking Technique

The key idea for the two-dimensional blocking technique is to move a small two-dimensional block through the grid updating all the nodes within the block. The block must be shaped as a parallelogram to obey all the data dependencies, and the update operations within the parallelogram can be performed in a line-wise manner from top to bottom. The principle of the technique will be described in the following by an example, which illustrates how all the nodes in the grid are updated twice during one global sweep. We start with the situation in Figure 6 where the blocking technique was used to create a valid initial state. Consider the left of the parallelograms at the left of the grid. Assume that the red and the black points in its lower half have already been updated once, while the upper part stays untouched so far. The number of updates performed on each node is represented by circles around that node. We now proceed to update all nodes in the parallelogram once. For this, we work in diagonals from top to bottom. As soon as the red points in the uppermost diagonal have been updated for the first time, the black points in the next diagonal within the parallelogram can also be updated for the first time. Then, the red and the black points on the diagonals underneath are updated. Note that for these this is now the second update. Finally, all the

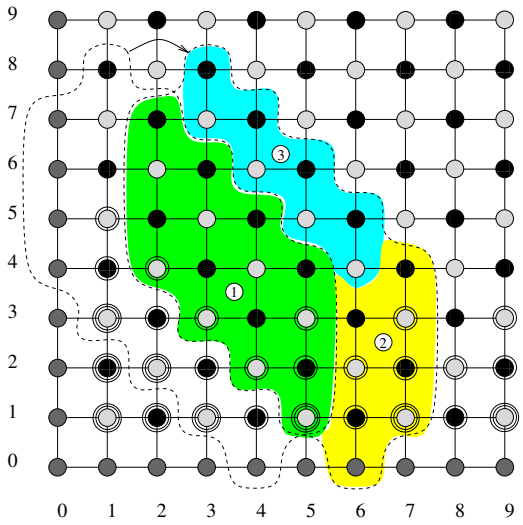


Figure 7: Data region classification for the two-dimensional blocking technique.

Relaxation Method	% of all accesses which are satisfied by				
	±	L1 Cache	L2 Cache	L3 Cache	Memory
Standard	5.1	27.8	50.0	9.9	7.2
Fusion	20.9	28.9	43.1	3.4	3.6
Blocking (2)	21.1	29.1	43.6	4.4	1.8
Blocking (3)	21.0	28.4	42.4	7.0	1.2
Wiper (4)	36.7	25.1	6.7	10.6	20.9
Wiper/p (4)	37.7	54.0	5.5	1.9	1.0

Table 3: Memory access behavior of different red-black Gauss-Seidel variants using a 1024×1024 grid.

points in the left parallelogram have been modified. For the right parallelogram this creates the same state as initially for the left parallelogram where the nodes on the two lower diagonals have already been updated once and the two upper diagonals are still untouched. Thus, we may switch to the right parallelogram and start updating again, using the same pattern. In that fashion the parallelogram is moved through the grid until it touches the right boundary. As soon as we reach the right boundary of the grid, we need to do some extra boundary handling and then move the parallelogram upward by four grid lines. There, some boundary handling on the left side follows, before we begin anew.

The data needed for an update of the grid points within one parallelogram are defined by all the nodes within a dotted drawn shape around the parallelogram (see Figure 6). The data values within the overlapping region of two adjacent dotted shapes (area 1 in Figure 7) are the data values which are directly reused between two parallelogram updates. The size of the data depends on the number of simultaneously performed update sweeps. Since one to four smoothing steps between the grid transfer operations are performed in typical multigrid algorithms, the size of that data should be small enough to fit at least in the L1 cache. In our example, the amount of data being reused directly is 128 Bytes (16×8 Bytes, area 1). Area 2 can be reused from a previous slide through the grid from left to right, as long as enough grid lines can be stored in one of the levels of the memory hierarchy. Typically, the data for area 2 is stored in one of the intermediate levels of the hierarchy. The rest of the data (area 3), however, must be fetched from main memory.

3.3 Performance Analysis

We continue with an analysis of the memory behavior of the optimized red-black Gauss-Seidel relaxation for a 1024×1024 grid using *DCPI* on a Digital PWS 500au. The column "Memory" of row "Fusion" of Table 3 shows that the *fusion technique* has succeeded in halving the number of main memory accesses compared to the standard red-black Gauss-Seidel algorithm.

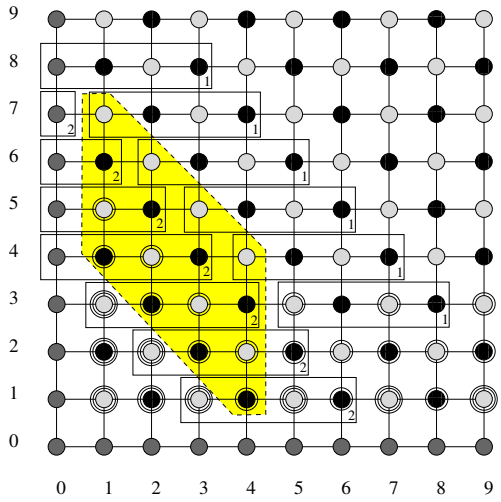


Figure 8: Alpha 21164 L1 cache mapping for a 1024×1024 grid.

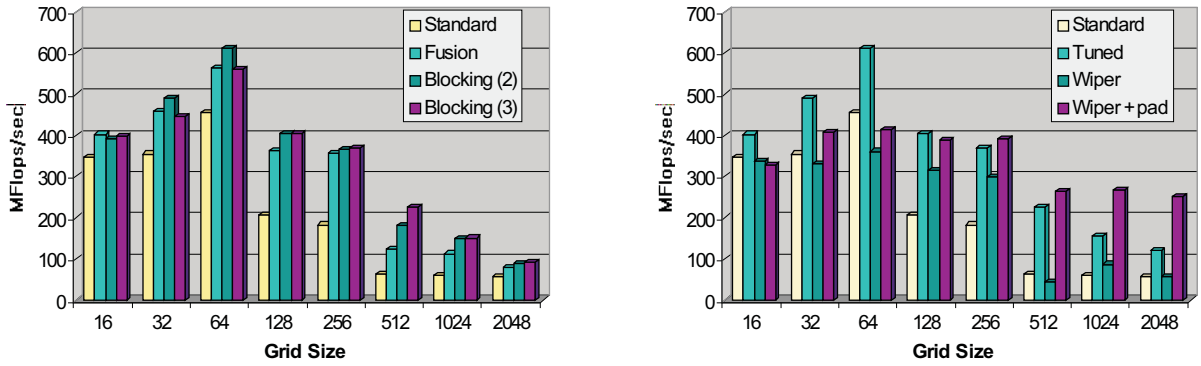


Figure 9: MFLOPS for different red-black Gauss-Seidel variants on a Digital PWS 500au.

An additional effect, which can improve the data locality using the fusion technique, is that the compiler might keep the values of the two updated nodes in a register, so that the update of the black node in row *i-1* saves two load operations. Since 14 memory operations are required for the two update operations (six loads and one store for each update) about 15 % of all memory operations can be saved through better register usage. The percentage of main memory accesses needed for red-black Gauss-Seidel using the blocking technique should be equal to the percentage of main memory accesses needed for the fusion technique divided by the number of blocked iterations. Row "Blocked (2)" and "Blocked (3)" of Table 3 show that the blocking technique does indeed reduce the number of memory accesses by a factor of two or three, respectively. The analysis of the memory behavior of the two-dimensional blocking technique[2] performing four successive red-black Gauss-Seidel sweeps simultaneously is shown in the row "Wiper (4)". Although the utilization of the registers improved slightly, the L2 and L3 cache utilization is worsened dramatically. Consequently, more than 20 % of all array references are not cached and therefore lead to main memory references.

The reason for this is a very high number of cross interference misses. A visualization with CVT [20] shows that throughout the whole run only four cache lines of the direct mapped L1 cache are used simultaneously. Surprisingly, even the L2 cache which is 3-way set associative in the case of the Digital Alpha 21164 processor cannot resolve the conflict misses. A possible mapping of the nodes within a parallelogram for a 1024×1024 grid on the cache lines of the Alpha 21164 L1

[2] for reasons of illustration called *wiper technique*

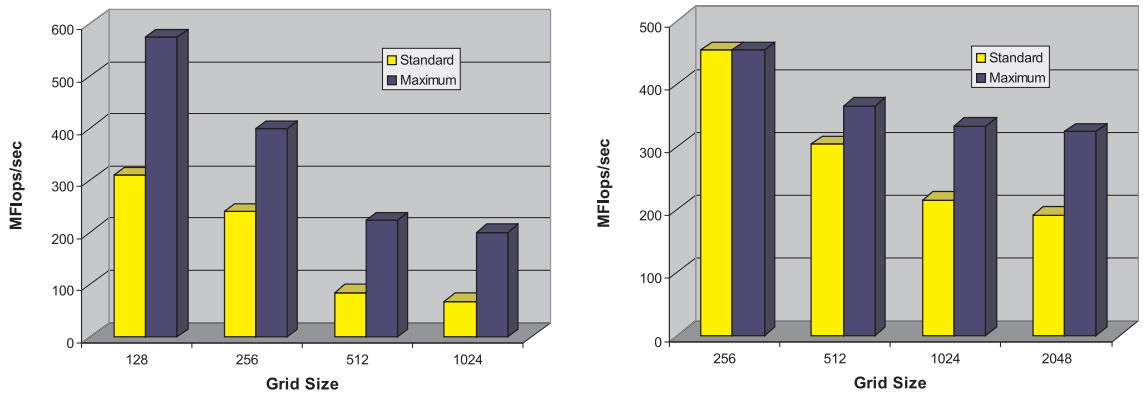


Figure 10: MFLOPS for a multigrid code on a Digital PWS500au (left side) and a Compaq XP1000 (right side).

cache is shown in Figure 8. All elements of the uppermost diagonal are mapped to the same cache line (cache line number 1). Hence, the data of the uppermost diagonal is not reused during the update of the second diagonal. Furthermore, the data needed for the update of a single node is producing a conflict in the L1 cache. For example, when updating the red node (1,7), the nodes (0,7), (1,8), (2,7), (1,6), and (1,7) are needed. Thus, accesses to the nodes (0,7) and (1,6), (1,8) and (2,7), as well as (1,8) and (1,7) are causing conflict misses.

A common technique to reduce the number of conflict misses is array padding [17, 18]. We used intra array padding for the two arrays of our relaxation algorithm. The size of the padding can be evaluated by the *INTRAPAD* algorithm described in [17]. Row "Wiper/p (4)" of Table 3 shows indeed that the two-dimensional blocking technique now utilizes the registers and the L1 cache much better than all other techniques. With this code, more than 90 % of all accesses are satisfied by the L1 cache or the registers.

The runtime performance of all techniques on a Digital PWS 500au is summarized in Figure 9. The fusion and blocking techniques can improve the performance for all grid sizes. For the small grids, which fit in the L2 cache, the improvement is only marginal. Nevertheless, the performance of 600 MFLOPS for a 64×64 grid is very impressive. The greatest speedup is achieved for the grids which do no longer fit in the L2 cache and for the 512×512 grid, which is the first grid that does no longer fit in the L3 cache. For the 512×512 grid the speedup is a factor of 3.6. However, none of the techniques can increase the performance of the algorithm for the very large grids. The reason for this is, that all techniques assume that a certain number of grid lines fit in the cache. For the 2048×2048 grid, however, less than six lines actually fit in the L2 cache. Therefore, the blocking technique is not effective anymore. Performing more update sweeps simultaneously in this case is counterproductive because the cache would have to hold even more data. Moreover, the goal of a high L1 cache utilization is still not reached. The performance of the two-dimensional blocking technique performing four update sweeps simultaneously is shown on the right side of Figure 9, compared to the standard and the maximum performance of fused and blocked ("Tuned") red-black Gauss-Seidel. The two-dimensional blocking technique without array padding is not performing satisfactorily. For the small grids and for the larger grids, the floating point performance is equal or even worse than the performance of the standard implementation. The padded version, however, achieves a remarkable speedup especially for the large grids. Figure 10 shows the speedups which can be obtained for a complete multigrid V-Cycle with four presmoothing and no postsmoothing steps using the red-black Gauss-Seidel smoother on a Digital PWS 500au, which is based on the Alpha 21164 chip, and a Compaq XP1000, which uses the successor chip Alpha 21264. Both machines run at a clock rate of 500 MHz and have a peak performance of 1 GFLOPS each. The XP1000 has a much faster memory system. Therefore, the achievable speedups are higher for the Alpha 21164-based architecture.

So far, we have shown that data locality optimization techniques can speed up a two-dimensional red-black Gauss-Seidel code on Alpha-based architectures. Figures 11 and 12 show the performance of a standard red-black Gauss-Seidel implementation (left sides) compared to the best possible performance obtained by the previously described optimizations (right sides) on several workstations for a 5-point discretization and a 9-point discretization of the differential operator, respectively[3]. Tables 4 and 5 show that the data locality optimization techniques are able to speed up the computation on all workstations.

[3]The 9-point stencil benchmark results for the HP PA8200 have to be considered preliminary due to some compiler instabilities during the optimization phase.

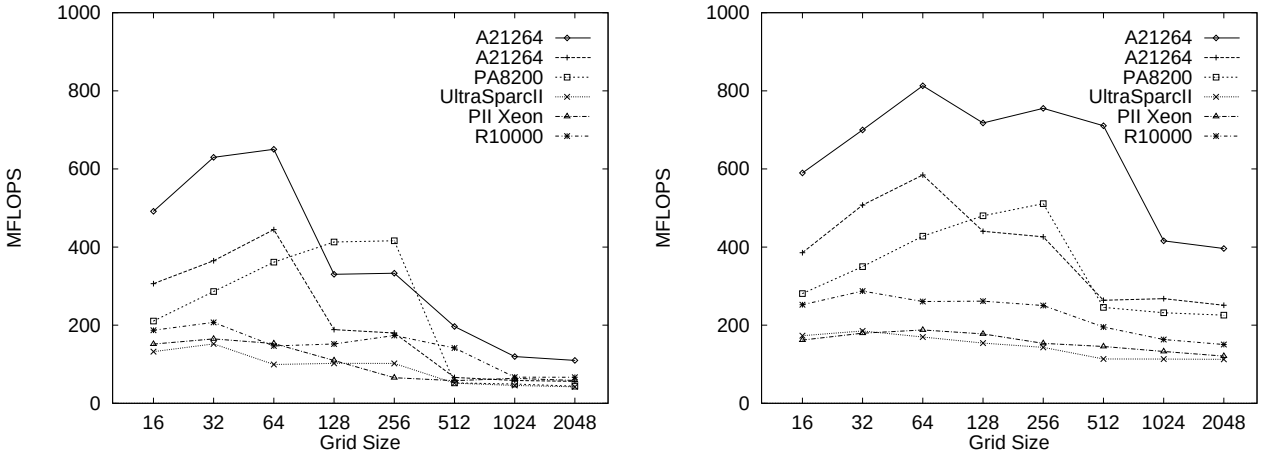


Figure 11: MFLOPS for a 5-point standard (left side) and an optimized (right side) red-black Gauss-Seidel method on several workstations.

Grid Size	SUN UltraSparc II			HP PA8200			SGI R10000		
	MFLOPS		Speedup	MFLOPS		Speedup	MFLOPS		Speedup
16	132.17	173.16	1.3	210.65	280.87	1.3	187.02	252.33	1.3
32	152.22	185.24	1.2	286.27	349.89	1.2	207.26	287.30	1.4
64	99.28	169.73	1.7	361.27	427.76	1.2	146.96	260.58	1.8
128	102.43	154.09	1.5	412.90	480.23	1.2	151.92	261.62	1.7
256	102.05	143.20	1.4	416.16	511.20	1.2	173.57	250.32	1.4
512	51.62	113.60	2.2	52.22	245.65	4.7	141.55	195.32	1.4
1024	45.60	113.40	2.5	49.25	231.73	4.7	66.90	163.21	2.4
2048	42.23	112.68	2.7	43.82	225.63	5.1	66.79	150.50	2.3
	Alpha 21164			Alpha 21264			Pentium II Xeon		
	MFLOPS		Speedup	MFLOPS		Speedup	MFLOPS		Speedup
16	306.40	385.51	1.3	491.52	589.82	1.2	151.63	162.04	1.1
32	365.10	507.65	1.4	629.80	699.76	1.1	164.87	179.60	1.1
64	444.64	584.52	1.3	650.28	812.85	1.3	153.01	187.70	1.2
128	188.76	440.43	2.3	330.32	717.74	2.2	109.56	177.75	1.6
256	179.96	426.15	2.4	332.93	755.18	2.3	65.60	153.40	2.3
512	66.35	263.76	4.0	196.61	710.62	3.6	57.93	145.39	2.5
1024	58.50	267.91	4.6	119.60	415.93	3.5	63.85	132.74	2.1
2048	56.22	250.92	4.5	109.91	396.37	3.6	58.50	120.76	2.1

Table 4: Speedups for a 5-point red-black Gauss-Seidel relaxation code on several workstations.

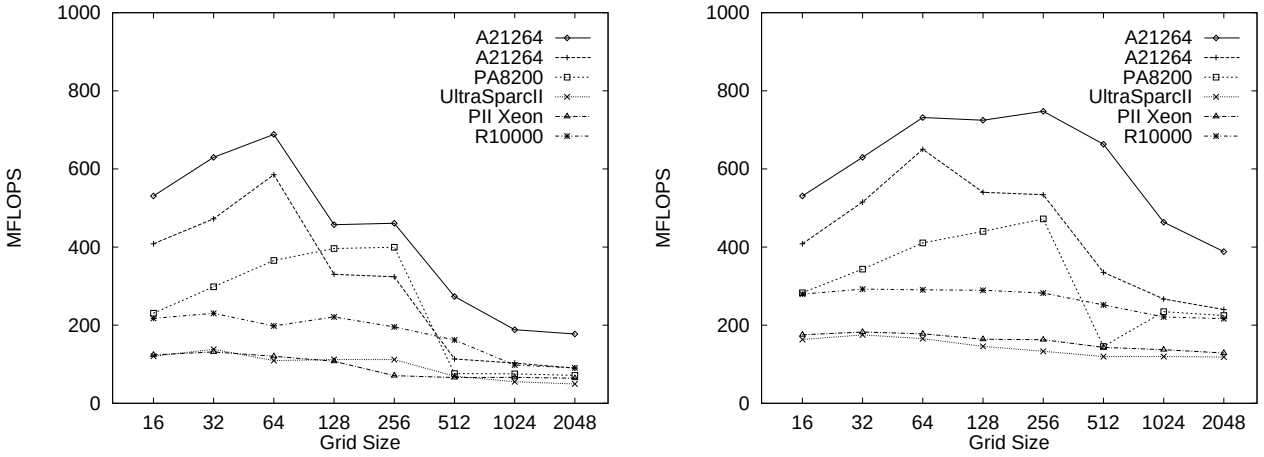


Figure 12: MFLOPS for a 9-point standard (left side) and an optimized (right side) red-black Gauss-Seidel method on several workstations.

Grid Size	SUN UltraSparc II			HP PA 8200			SGI R10000		
	MFLOPS		Speedup	MFLOPS		Speedup	MFLOPS		Speedup
16	120.65	163.34	1.4	230.80	283.11	1.3	217.56	279.57	1.3
32	138.25	175.30	1.3	298.33	343.52	1.2	230.41	292.47	1.3
64	109.39	165.62	1.5	365.78	410.65	1.1	198.06	290.56	1.5
128	112.19	145.84	1.3	396.39	440.21	1.1	221.03	289.47	1.3
256	112.01	133.17	1.2	399.51	472.18	1.2	195.52	282.35	1.4
512	69.55	119.68	1.7	76.15	145.59	1.9	162.16	251.87	1.6
1024	55.30	119.79	2.1	75.35	234.52	3.1	98.10	221.03	2.3
2048	49.66	118.64	2.4	71.41	224.67	3.1	90.74	216.73	2.4
Grid Size	Alpha A21164			Alpha 21264			Pentium II Xeon		
	MFLOPS		Speedup	MFLOPS		Speedup	MFLOPS		Speedup
16	408.34	408.34	1.0	530.84	530.84	1.0	124.03	175.37	1.4
32	472.35	515.29	1.1	629.80	629.80	1.0	131.51	182.55	1.4
64	585.25	650.28	1.1	688.53	731.57	1.1	120.42	178.00	1.5
128	330.32	540.26	1.6	457.37	724.74	1.6	107.52	163.94	1.5
256	323.93	534.02	1.6	460.98	747.63	1.6	70.84	162.80	2.3
512	113.51	335.35	3.0	273.47	663.25	2.4	65.97	143.25	2.2
1024	103.04	266.96	2.6	188.38	463.69	2.5	66.50	137.31	2.1
2048	90.06	239.99	2.7	177.47	388.21	2.2	64.26	128.85	2.0

Table 5: Speedups for a 9-point red-black Gauss-Seidel relaxation code on several workstations.

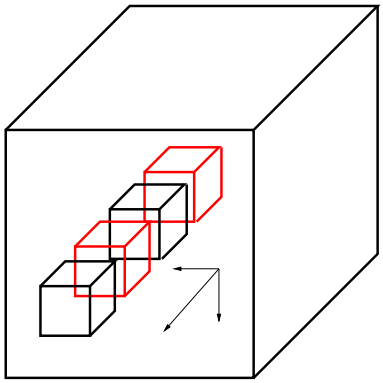


Figure 13: Three-dimensional blocking technique for red-black Gauss-Seidel.

4 Cache Optimization for Red-Black Gauss-Seidel in 3D

The performance results for a standard 7-point implementation of a red-black Gauss-Seidel smoother in three dimensions are comparable to the 2D case. Especially for larger grids, the MFLOPS rates drop dramatically on a wide range of currently available machines. Again, this is due to the fact, that data cannot be cached between successive smoothing iterations.

To overcome this effect, we propose a three-dimensional blocking technique, which is illustrated in Figure 13. This technique uses a cuboid of nodes which is relatively small compared to the original three-dimensional grid. This cuboid is moved through the original grid, starting in its bottom south-east corner. In order that all the data dependencies of the red-black Gauss-Seidel method are still being respected our blocking algorithm has to be designed in the following manner.

After all the red points inside the working set defined by the current position of the cuboid have been relaxed, it must be re-positioned within the original grid: it has to be moved one step towards the left, one step towards the front and one step towards the bottom boundary plane of the grid. Of course, the cuboid cannot exceed the boundary planes of the grid. Therefore, we must implement special routines for the grid planes that are located close to the boundaries of the original grid structure.

After re-positioning the cuboid, the black points inside its new working set can be updated, before the cuboid is again moved on to its next position. If several red-black Gauss-Seidel iterations are to be blocked, this next position is obtained by again moving the cuboid one step in each space dimension. If, however, only one sweep over the red points and one sweep over the black points are to be fused together, the subsequent position of the cuboid is next to the position it had before re-positioning it in order to update the black grid nodes.

It is apparent that this algorithm incorporates the techniques of loop fusion and loop blocking, which have been described in detail for the two-dimensional case in Section 3 and can easily be generalized to the case of three dimensions.

The four positions of the cuboid shown in Figure 13 illustrate that two successive red-black Gauss-Seidel iterations have been blocked into one single sweep through the whole grid.

However, this technique by itself does not lead to significant speedups on the A21164-based PWS 500au, for example. A closer look at the cache statistics using *DCPI* shows that the poor performance is caused by a high rate of cache conflict misses. As we have mentioned in Section 3, these effects also occur in the case of two array dimensions.

In the three-dimensional case, the occurrence of cache conflict misses can also be illustrated easily. We assume a grid containing 64^3 double precision values which occupy 8 Bytes of memory each. Furthermore, we assume a direct-mapped cache the size of which is 8 KByte (such as the L1 cache of the A21164 processor). Consequently, every two grid points which are adjacent in the trailing array dimension map to the same cache line and therefore cause each other to be evicted from the cache, in the end resulting in a poor performance of the code.

Again, as in the two-dimensional case, array padding is an appropriate technique to reduce this effect. Figure 14 shows two adjacent planes of the original grid in order to illustrate our padding strategy. Firstly, we introduce padding in x-direction in order to avoid cache conflict misses caused by grid points which are adjacent in dimension y. Secondly, we use padding to increase the distance between neighboring planes of the grid. This kind of padding is illustrated by the shaded box in Figure 14. It reduces the effect of z-adjacent nodes causing cache conflicts. Particularly this kind of inter-plane padding is crucial for code efficiency and has to be implemented carefully. It has to be seen as a non-standard padding technique

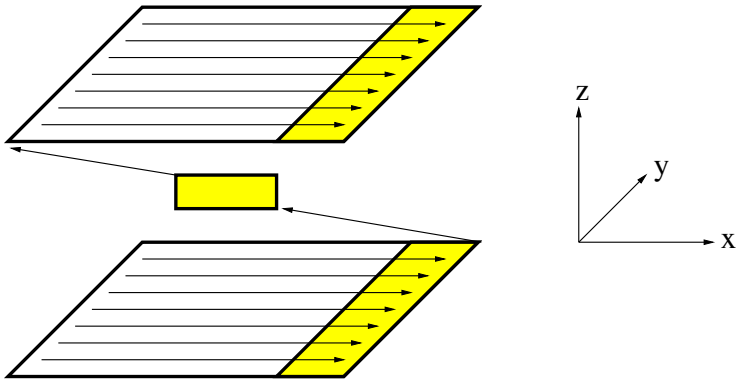


Figure 14: Padding technique for three-dimensional grids.

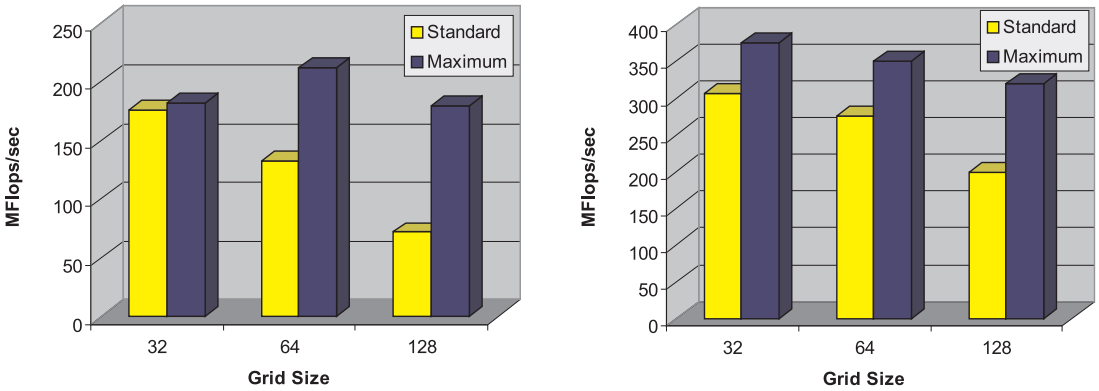


Figure 15: MFLOPS for a 3D red-black Gauss-Seidel algorithm on a Digital PWS500au (left side) and a Compaq XP1000 (right side).

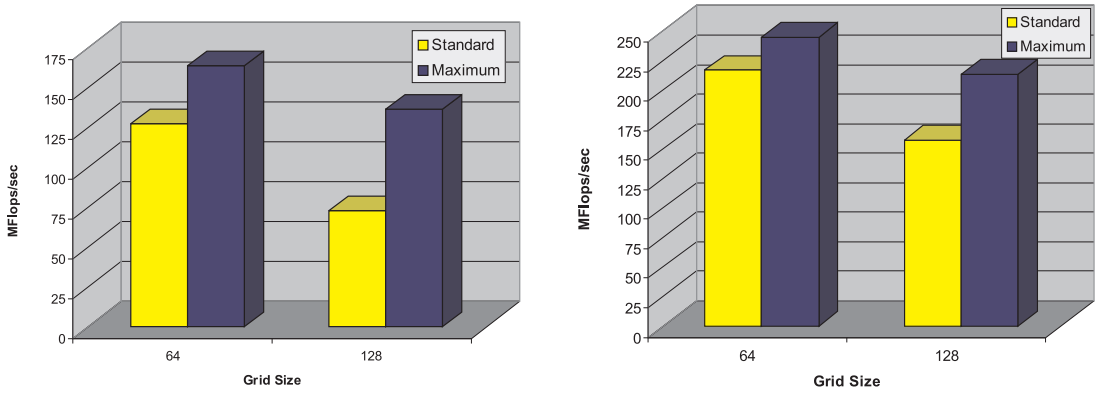


Figure 16: MFLOPS for a 3D multigrid code on a Digital PWS500au (left side) and a Compaq XP1000 (right side).

which consumes less memory than the standard ones, which are based on simply extending the three-dimensional arrays in two dimensions. Figures 15 and 16 show what speedups can be obtained on the A21164-based Digital PWS 500au and the A21264-based Compaq PWS 500au workstations. The optimization techniques for the three-dimensional case are not restricted to one certain architecture. Similar speedups can be shown for other architectures as well. We only consider grids with 64^3 and 128^3 nodes, since the methods are not effective when applied to smaller problems.

5 Conclusions

We have outlined the cache behavior of iterative methods using as a simple example the method of Gauss-Seidel with a red-black ordering of the grid nodes. Besides, we have demonstrated how cache optimization techniques can be applied to iterative methods in both two and three dimensions. Their effectiveness has been shown by extensively describing runtime and profiling experiments.

Iterative methods repeatedly perform global sweeps through data structures which are typically much larger than the L1, L2 or even L3 caches of current machines. Consequently, the efficiency of caches is dramatically reduced by capacity misses, and the performance of iterative methods is bound by the speed of main memory accesses. Repeated Gauss-Seidel sweeps, as they are used for example in the smoothing part of a multigrid algorithm, can be restructured in order to obtain high cache reuses out of rather small working sets. The two-dimensional blocking technique can be optimized for caches as small as the 8 KByte L1 cache, e.g. found in the Alpha 21164 chip, and the primary working set size is even independent of the grid size. This is accomplished by a carefully designed reorganization of the processing order, combined with array padding to avoid cache associativity conflicts. For the three-dimensional case, comparable techniques can be employed.

We have restricted ourselves to optimizations which adhere to all data dependencies, so that, in principle, these code transformations are quite simple. Still, most of the compilers presently cannot apply them even to elementary iterative methods, like for example the Gauss-Seidel method. Our experimental results show that, although our codes have already been compiled with aggressive optimizations enabled, these techniques can still speed up iterative methods on all modern platforms by a factor of two to five depending on the relative speed of the processor and the memory architecture.

Our future work will focus on more general algorithms, including variable coefficient problems and more general grids. In particular, this means that the amount of data associated with each point of the grid is larger than in the case of constant coefficients, which occur when solving Poisson's equation on an equidistant grid, for example. More data per point may imply that it does not make sense to optimize for the small caches of the memory hierarchy, but only for the larger ones, i.e. the ones which have at least several hundred kilobytes in size. Besides, the investigation of the cache friendly treatment of adaptively refined structured grids and even completely unstructured meshes — e.g. arising in the context of finite element discretizations of partial differential equations — has only recently begun [7, 14]. In all these more general cases, it is not the grid vector of the unknowns but the coefficients of the sparse matrix of the resulting linear system of equations which determines the cache behavior and eventually the performance of the code, since this is the data structure which consumes most of the memory.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorenson. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1992.

[2] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 1–14, St. Malo, France, October 1997.

[3] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345ff, December 1994.

[4] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology. In *Proceedings of International Conference on Supercomputing*, July 1997.

[5] A. Brandt. Multigrid Techniques: 1984 Guide with Applications to Fluid Dynamics. *GMD Studien*, 85, 1984.

[6] C. C. Douglas. Caching in With Multigrid Algorithms: Problems in Two Dimensions. *Parallel Algorithms and Applications*, 9:195–204, 1996.

[7] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß. Cache Optimization for Structured and Unstructured Grid Multigrid. *Submitted to Electronic Transactions on Numerical Analysis (ETNA)*, 1999.

[8] D. Quinlan F. Bassetti, K. Davis. Temporal Locality Optimizations for Stencil Operations within Parallel Object-Oriented Scientific Frameworks on Cache-Based Architectures. In *Proceedings of the PDCS'98 Conference*, Las Vegas, Nevada, July 1998.

[9] M. Frigo and S. G. Johnson. The Fastest Fourier Transform in the West. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology, September 1997.

[10] L. Gwennap. Digital 21264 Sets New Standard. *Microprocessor Report*, 10(14), October 1996.

[11] W. Hackbusch. *Multigrid Methods and Applications*. Springer Verlag, Berlin, 1985.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.

[13] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving Locality Using Loop and Data Transformations in an Integrated Framework. In *Proc. 31st International Symposium on Micro-Architecture (MICRO-31)*, Dallas, Texas, December 1998.

[14] D.K. Kaushik, D.E. Keyes, and B.F. Smith. On the Interaction of Architecture and Algorithm in the Domain-Based Parallelization of an Unstructured Grid Incompressible Flow Code. In J. Mandel et al., editor, *Proceedings of the 10th Intl. Conf. on Domain Decomposition Methods*, pages 311–319, 1998.

[15] D. Loshin. *Efficient Memory Programming*. McGraw-Hill, New York, NY, 1999.

[16] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, December 1995.

[17] G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, Montreal, Canada, June 1998.

[18] G. Rivera and C.-W. Tseng. Eliminating Conflict Misses for High Performance Architectures. In *Proceedings of the 1998 International Conference on Supercomputing (ICS'98)*, Melbourne, Australia, July 1998.

[19] L. Stals, U. Rüde, C. Weiß, and H. Hellwagner. Data Local Iterative Methods for the Efficient Solution of Partial Differential Equations. In *Proceedings of the The Eighth Biennial Computational Techniques and Applications Conference*, Adelaide, Australia, September 1997.

[20] E. van der Deijl, O. Temam, E. Granston, and G. Kanbier. The Cache Visualization Tool. *IEEE Computer*, July 1997.

[21] 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*, June 1991.