FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Cache optimizations for multigrid in 3D

Nils Thürey

Studienarbeit

Cache optimizations for multigrid in 3D

Nils Thürey

Studienarbeit

Aufgabensteller:	Prof. Dr. Ulrich Rüde
Betreuer:	Dipl.–Inf. Markus Kowarschik
Bearbeitungszeitraum:	2001-12-01 bis 2002-06-12

Erklärung:

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 12. Juni 2002 ....

Contents

Abstract

To obtain the numerical solution for *partial differential equations* (PDEs) it has been shown that multigrid methods work efficiently for these problems, as they perform with an asymptotically linear complexity. Still, for this class of algorithms, a limiting factor is the execution speed, which is currently determined by the access to the main memory. However, the speed of computer processors is currently increasing much faster than this memory latency decreases. Thus, modern computer architectures employ memory hierarchies, that are 30 to 40 times faster than the main memory [HP96], to mitigate this effect. It has been shown by [WKKR99, Pfä00, Wei01] that multigrid methods in 2D can be optimized in terms of spatial and temporal locality, to efficiently use these memory hierarchies. In this thesis, the transferability of these techniques to 3D problems will be investigated. Besides quantitative values for the gain in performance, it will be explained, why these optimizations yield results different to the 2D case, and where the inherent limitations are.

Chapter 1

Introduction

1.1 Multigrid in 3D

Multigrid methods are among the most efficient algorithms for the solution of elliptic partial differential equations (PDEs), and as such play an important role in many technical and physical applications. This section will give a brief overview, see [BHM00] and [TOS01] for a detailed description of multigrid methods.

Systems of linear equations can be solved by a direct solver (e.g. Gaussian elimination), but for different reasons like memory requirements and high complexity these solvers often cannot be applied to larger problems anymore. Another option is to use iterative methods, the most basic of which are the *Jacobi Method* or the *Gauss-Seidel Method*.

The numerical solution of PDEs often involves the solution of systems of linear equations of the form

$$Ax = b \tag{1.1}$$

A Gauss-Seidel step then requires the following calculation for all unknowns

$$u_j^{(n+1)} = \sum_{i=1}^{j-1} a_{ij} u_i^{(n)} + \sum_{i=j+1}^{n} a_{ij} u_i^{(n+1)}, \quad 1 \leq j \leq n \tag{1.2}$$

But, due to the effect that Gauss-Seidel and similar iterative solvers are inefficient for already smooth approximations, multigrid methods exploit the fact, that lower frequencies appear as higher frequencies on coarser grids. Thus the problem is solved using different levels of discretization, but instead of solving the original equations, the residual is calculated as

$$r = Ax - f, \tag{1.3}$$

and restricted to a coarser grid. The residual equation is then solved

$$Ar = e, \tag{1.4}$$

after which the correction of the approximation is prolongated to the finer grid again. This residual equation has the same form as Equation (1.1), thus this is a recursive process.

In the following, the focus will be on problems described by the scalar elliptic equation

$$\nabla \cdot (a \nabla u) = f \tag{1.5}$$

The resulting Gauss-Seidel iteration corresponding to the equation for unknown $u_{x,y,z}$ is

$$\begin{aligned} u_{x,y,z} = 1/c_{x,y,z}(f \quad &- \quad w_{x,y,z}u_{x-1,y,z} - e_{x,y,z}u_{x+1,y,z} \\ &- \quad s_{x,y,z}u_{x,y-1,z} - n_{x,y,z}u_{x,y+1,z} \\ &- \quad b_{x,y,z}u_{x,y,z-1} - t_{x,y,z}u_{x,y,z+1}) \end{aligned} \tag{1.6}$$

where w, e, s, n, b, t are the coefficients for the west, east, north, south, lower and upper neighbors, respectively, and c represents the coefficient of the current point $u_{x,y,z}$.

For the implementation, only Dirichlet boundary conditions of the form

$$u = \phi \text{ on } \partial\Omega$$

are used. The restriction of the residual is done using full-weighting. Let $I_h^{2h} v^h = v^{2h}$,

$$\begin{aligned}
v^{2h}_{x,y,z} = \frac{1}{64}[\quad & v^h_{2x-1,2y-1,2z-1} + v^h_{2x+1,2y-1,2z-1} + v^h_{2x-1,2y+1,2z-1} + v^h_{2x+1,2y+1,2z-1} \\
& + v^h_{2x-1,2y-1,2z+1} + v^h_{2x+1,2y-1,2z+1} + v^h_{2x-1,2y+1,2z+1} + v^h_{2x+1,2y+1,2z+1} \\
& + 2\ (v^h_{2x-1,2y-1,2z} + v^h_{2x+1,2y-1,2z} + v^h_{2x-1,2y+1,2z} \\
& + v^h_{2x+1,2y+1,2z} + v^h_{2x-1,2y,2z-1} + v^h_{2x+1,2y,2z-1} \\
& + v^h_{2x-1,2y,2z+1} + v^h_{2x+1,2y,2z+1} + v^h_{2x,2y-1,2z-1} \\
& + v^h_{2x,2y+1,2z-1} + v^h_{2x,2y-1,2z+1} + v^h_{2x,2y+1,2z+1}) \\
& + 4\ (v^h_{2x-1,2y,2z} + v^h_{2x+1,2y,2z} + v^h_{2x,2y-1,2z} \\
& + v^h_{2x,2y+1,2z} + v^h_{2x,2y,2z-1} + v^h_{2x,2y,2z+1}) \\
& + 8\ v^h_{2x,2y,2z}\ ], \qquad 1 \le x, y, z \le \frac{n}{2} - 1
\end{aligned} \tag{1.7}$$

The error is prolongated to the fine grid with trilinear interpolation, letting $I_{2h}^h v^{2h} = v^h$,

$$\begin{aligned}
v^h_{2x,2y,2z} &= v^{2h}_{x,y,z} \\
v^h_{2x+1,2y,2z} &= \frac{1}{2}(v^{2h}_{x,y,z} + v^{2h}_{x+1,y,z}) \\
v^h_{2x,2y+1,2z} &= \frac{1}{2}(v^{2h}_{x,y,z} + v^{2h}_{x,y+1,z}) \\
v^h_{2x,2y,2z+1} &= \frac{1}{2}(v^{2h}_{x,y,z} + v^{2h}_{x,y,z+1}) \\
v^h_{2x+1,2y+1,2z} &= \frac{1}{4}(v^{2h}_{x,y,z} + v^{2h}_{x+1,y,z} + v^{2h}_{x,y+1,z} + v^{2h}_{x+1,y+1,z}) \\
v^h_{2x,2y+1,2z+1} &= \frac{1}{4}(v^{2h}_{x,y,z} + v^{2h}_{x,y+1,z} + v^{2h}_{x,y,z+1} + v^{2h}_{x,y+1,z+1}) \\
v^h_{2x+1,2y,2z+1} &= \frac{1}{4}(v^{2h}_{x,y,z} + v^{2h}_{x+1,y,z} + v^{2h}_{x,y,z+1} + v^{2h}_{x+1,y,z+1}) \\
v^h_{2x+1,2y+1,2z+1} &= \frac{1}{8}(v^{2h}_{x,y,z} + v^{2h}_{x+1,y,z} + v^{2h}_{x,y+1,z} + v^{2h}_{x,y,z+1} \\
&\quad + v^{2h}_{x+1,y+1,z} + v^{2h}_{x,y+1,z+1} + v^{2h}_{x+1,y,z+1} \\
&\quad + v^{2h}_{x+1,y+1,z+1}), \qquad 0 \le x, y, z \le \frac{n}{2} - 1
\end{aligned} \tag{1.8}$$

Due to the iterative nature of the multigrid algorithm, i.e. the successive traversal of the data set until a given convergence criteria has been fulfilled, they exhibit a high degree of temporal locality. This motivates the effort to cache optimize multigrid implementations. Our multigrid program, which will be described in more detail in Chapter 2, uses a *red-black* ordering of the unknowns, as the data dependencies then allow the update of red or black points in any order. As multigrid scheme, a standard V-cycle will be used. In [Yav96], it is shown that a red-black SOR is more cost-effective in 3D, and usually yields better convergence results. All optimization techniques discussed in this thesis can be used for both red-black Gauss-Seidel as well as SOR.

1.2 Hardware Layout

Nowadays, while the speed of the CPU grows with rates of about eighty percent per year, the memory access times only decrease five to ten percent in the same period of time. To lessen the

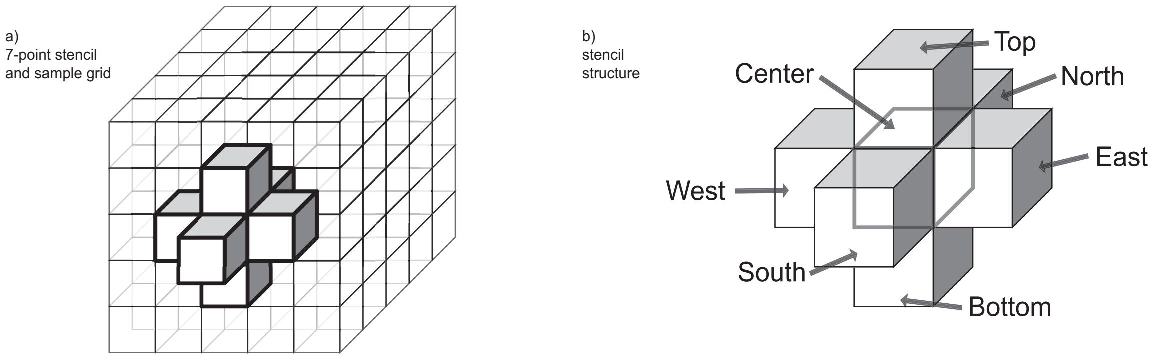

Figure 1.1: Stencil resulting from the described discretization.

effect of this growing disparity, hierarchies of caches are used in modern computer architectures [HP96, Han98]. Thus it is very important for the software to efficiently make use of all caches in a computer. Accesses to an element of data held in cache is an order of magnitude faster than the main memory access, which is the result of a *cache miss*, meaning that the needed data is currently not in the cache. The data is loaded into the cache levels in *cache lines* of usually 32 or 64 bytes.

To achieve the necessary speed, cache memories are much smaller than the main memory. The address of a cache line in the cache is determined with different strategies. *Direct-mapped caches* can store a data element only at a single address, determined for example by ($X \bmod C$), where X is the virtual address and C the size of the cache. It is assumed, that the virtual address is used to determine the cache address. The result of this is, that a subsequent access to the memory address $X + n * C$ will evict the element from address X from cache. Hence, the next access to the data in X will result in a cache miss. This can become a problem, if for example, two neighboring elements of a multidimensional array are stored at addresses with the distance being a multiple of the cache size, and these elements are always needed together in a calculation. The result is *cache-thrashing*, and a significant drop in performance. The best way to avoid this problem, would be to store a cache line at any address in the cache. This is called a *fully-associative cache*. As this is currently unrealizable, due to the fact that for retrieving the element, the whole cache address space has to be searched, modern architectures employ *set-associative caches*. This enables a data element to be stored in N different places, for an N-way set associative cache.

Cache misses can be classified into the following three types. *Compulsory misses* occur, when a data item is accessed for the first time, and therefore cannot be in the cache yet. *Capacity misses* on the other hand, are caused by limited cache capacity. The data which is requested does not reside in cache anymore, as, between the two subsequent accesses, too many other data items have been accessed. If these other accesses are less than the total cache capacity, the miss results from the limited cache associativity, and would not have occurred in a fully-associative cache. In this case it is called a *conflict miss* [Los98].

If a code accesses data items which are stored close to each other, it exhibits *spatial locality*. A single cache line could for example satisfy the accesses to multiple floating point numbers with a single main memory access, before it is evicted again. *Temporal locality* on the other hand, means that the same data element is accessed again, once or more, after a short period of time.

For *instruction caches*, which are the pendant of the data caches, the size is more important than the cache address of an instruction. If a loop body cannot fit into the instruction cache, it will induce main memory accesses even if the data accesses can be completely satisfied from the data cache. This is especially important when unrolling more complex loop kernels, e.g. those from Section 4.6 and 4.7. Branching operations can also be responsible for instruction cache misses, if the result of the branch is not correctly predicted by the processor. Conditional statements in loop nests should for example be avoided at all costs [DS98]

Another cache memory is the *Translation Lookaside Buffer* (TLB). To support multi-tasking in hardware, the CPU has to resolve a virtual address for a running process to retrieve the physical address. The table to translate the virtual to physical addresses is also stored in main memory, and to avoid these main memory accesses modern architectures store 64 to 128 translations in the TLB for a faster lookup. If different data elements, spread too far over the virtual address space, are accessed, this can result in a main memory access to the page table although the data item itself is still stored in the cache. If the address of a page is not in the TLB, this is, of course, called *TLB miss*.

The cache optimizations proposed in this thesis were tested on different platforms, which are described in AppendixA on page 34. On Platform A, which uses an Alpha 21164 CPU, the profiling tool DCPI (Digital Continuous Profiling Infrastructure, see [ABD+97]) was used to obtain a detailed analysis of the multigrid program performance.

1.3 Cache Optimizations

Even though there are efforts to enhance the caches in hardware, like increasing the associativity or prefetching, the software also has to be optimized to improve the hit rates of caches. This is usually done by increasing the temporal and spatial locality, without changing the semantics of the program.

The validity of these transformations first has to be checked by *dependence analysis*, to assert that they do not violate any control or data dependencies of the program. Control dependencies arise from branching instructions, which execute statements depending on a certain condition. The flow of data in a program determines the data dependencies between the instructions. Dependence graphs or various dependence testing algorithms can be used, which are especially important for automated optimization in compilers.

The cache optimizations themselves concern either the data layout, or the data accesses. Data layout optimizations improve the locality of a program by reordering the data in a way, that either reduces conflict or TLB misses, or increases the cache or register reuse. The data for numerical codes are usually stored in relatively large arrays. If several types of values are needed, it has to be decided, whether these should be stored in separate arrays, or some of these arrays should be merged. The different possibilities for a variable coefficient multigrid program will be evaluated in Section 3.1. Generally, the data structure should reflect the access pattern of the implementation. If several data elements are always needed together in one calculation, an array merging could result in increased cache line reuse. When one data item of a cache line is accessed, all other values that are stored in the same cache line will not require further main memory accesses, as the cache line will always be loaded into cache as a whole.

Still, multiple accesses to a single multidimensional array can lead to conflict misses, especially when the array dimensions are multiples of the cache size. To avoid these negative effects, the arrays can be artificially enlarged. This is called *array padding*, and is usually done by increasing the array dimensions, without accessing these additional data buffers. This standard padding for a 3D array is illustrated in Figure 1.2, on the left side. Increasing the array in X-direction can reduce conflicts that arise when accessing data items in the same plane, while padding in Y-direction, *intra plane padding*, can prevent conflicts from accesses to multiple planes respectively. This standard padding, however, can lead to huge memory overhead, especially in the 3D case. As only the offset of virtual addresses in the cache is of importance, it is preferable to insert an arbitrary sized padding between two planes. This *non-standard padding* is displayed on the right of Figure 1.2. As can be easily seen, it is a superset of the standard padding sizes, as, given a standard padding of P_y, the non-standard padding can be set to $P_y * (D_x + P_x)$, where D_x and P_x are the size of the array in X-direction, and the padding in X-direction. Still, even with this non-standard padding, it is only possible to enlarge the array by multiples of the size of a single data item.

Other cache optimizations change the access order of the program. *Loop interchange* can be used to improve the stride of a computation. When the loops of a loop nest can be permutated, this should be done in a way that accesses the data according to the memory layout. The correct loop order can reduce the stride of the loop, improving the reuse in cache lines, and shorten the time between two subsequent accesses to a data item. For several loop nests that have the same iteration space, these can also be *fused*. The loop bodies are then executed together in a single loop nest, reducing the loop overhead and increasing the instruction level parallelism. Furthermore, when the data that is accessed does not fit in cache, each loop nest will result in new cache misses, while the same data can be immediately reused in a fused loop.

For multiply nested loops, the time between two subsequent accesses may be very long. Instead of working on the whole problem space, *loop blocking* (also called tiling) can be imagined as partitioning

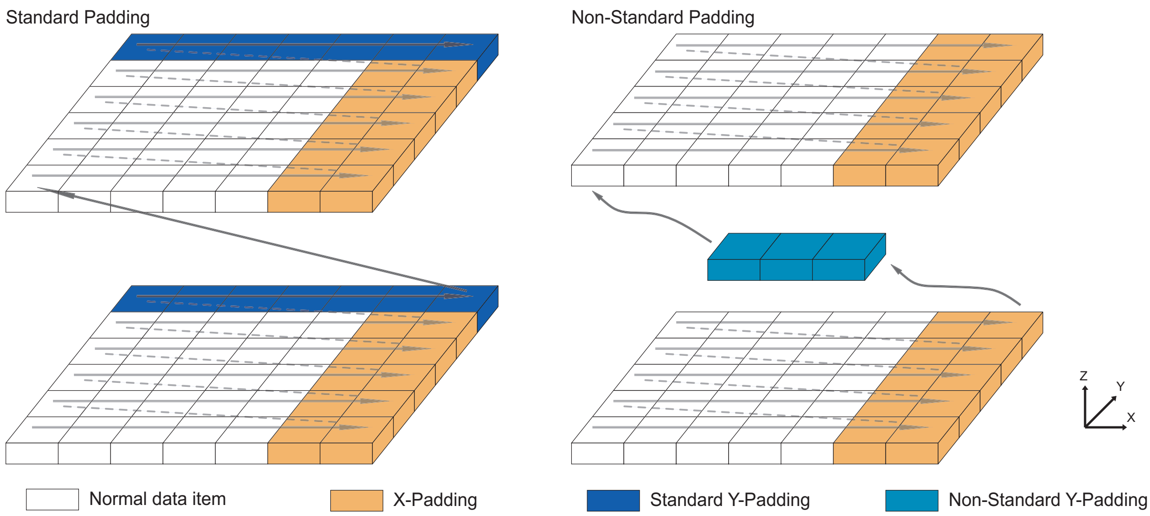

Figure 1.2: Standard padding can be seen on the left, non-standard padding, which introduces a certain number of data items, not necessarily a multiple of the y-dimension, on the right.

this space into smaller regions (*tiles*), that should be adapted to the size of the cache. Loop blocking is done by introducing additional loops, that run over a smaller set of values than the original ones. The drawback of this is the increased loop overhead and decreased pipeline performance due to the shorter loop lengths. When the loops get too short, basic compiler optimizations like software pipelining and unrolling are often not performed. On the other hand, the increased temporal locality can significantly speed up the program due to the reduced main memory accesses. A popular example for this optimization is matrix-matrix multiplication. A standard implementation is shown in Figure 1.3a, that is blocked in Figure 1.3b. While in the standard version, in between two accesses to $C(I, J)$, the complete array with its $N * N$ values has been accessed, the blocked version accesses only $BI * BJ$ data elements.

The application of loop transformations to general problem classes is a current research topic [SL99, GJF01], and various models have been developed to determine the transformation effects [MLW91]. Still, many of these techniques are often too complicated to be automatically applied by a compiler, and thus have to be manually implemented.

The optimizations in this thesis will focus on multigrid algorithms on structured grids [BDQ98], while other works like [DHK$^+$00] and [GKKS01] also include unstructured meshes, or other other algorithms like fast fourier transformations [FJ98] or bit-reversals [ZZ99]. Various optimization techniques have been proposed for multigrid in 2D [Dou96, Pfä00] and their applicability to the 3D case will be evaluated in the following chapters.

a)

```
do K=1, N
  do J=1, N
    SCALE = B(K,J)
    do I=1, N
      C(I,J) = C(I,J) + A(I,K)*SCALE
    enddo
  enddo
enddo
```

b)

```
do JJ=1, N, BJ
  do II=1, N, BI

    do K=1, N
      do J=JJ, min(JJ+BJ, N)
        SCALE = B(K,J)
        do I=II, min(II+BI, N)
          C(I,J) = C(I,J) + A(I,K)*SCALE
        enddo
      enddo
    enddo

  enddo
enddo
```

Figure 1.3: a) Standard matrix-matrix multiplication implementation b) Blocked version of the same code, two new loops are introduced, blocking the X and Y loop of a) by BI and BJ.

Chapter 2

Reference Results

This chapter describes the first experiments, which determine the possible speed of a multigrid program with no negative cache effects, and the effect of different loop orders in a red-black Gauss-Seidel implementation.

2.1 Determination of upper performance limits

To evaluate the possible effect of cache-optimizations for a 3D multigrid program, and to furthermore ensure that no pipeline-stalls or other similar effect spoil the cache-optimizations, the following experiments were performed. Several small test programs were written, which show the maximum performance of a red-black Gauss-Seidel kernel without any negative cache-effects. This requires that all data that is needed for the calculation has to reside without conflicts in the highest cache-level. To compare the performance for different problems, all programs were made in three variants. One to only solve homogenous problems without a right hand side or coefficients, a second introducing a right hand side for inhomogeneous problems, and, corresponding to the multigrid program used in the following chapters, a red-black Gauss-Seidel implementation with right hand side, and seven coefficients per unknown. The first variant thus uses one double per unknown, resulting in 8 bytes of memory required per grid point. The second variant uses 16 bytes for the two doubles, while the third requires 9 floating point values, and a total of 72 bytes per unknown. To ensure that no cache-thrashing appears, these values are stored in an equation-oriented data layout (see Section 3.1). To enable as many compiler optimizations as possible, the arrays were passed to the Gauss-Seidel method to make the compiler aware of the 3D structure, as will be described in Section 3.2.

2.1.1 Test Programs

For each of these variants there are six programs. All of which, except the last one, run on 33^3 grids. This sixth program, which should show the full effect of problems too big to fit in cache, runs on a 129^3 grid. While this last program performs a complete red-black Gauss-Seidel step, the other programs use Gauss-Seidel-like computational kernels with similar calculations. All programs are padded to avoid pathological cases.

1. The first test runs through a single line of points in the 33^3 grid, relaxing only the 16 red points over and over again. This was used as a reference with minimum memory requirements, and minimum loop overhead, as only a single iteration loop and an x-loop are necessary.

2. The next step is to perform a red black relaxation. In the iteration loop there are now two loops, one running over the red, the other running over the black points. Now there is a total of 31 inner points which are relaxed.

3. Similar to the optimization described in Section 4.3 the third test program has fused loops , relaxing a red point `(x,y,z)` and then its black neighbor `(x,y,z-1)`, in the adjacent plane below. This again reduces the program structure to an iteration loop, and a single loop over the X-coordinates, relaxing a total of 32 unknowns.

4. The impact of the introduction of the two other loops over the Y- and Z-coordinates is tested in the next program. These two loops are trivial and only run over one value, thus should not result in a significant change in performance with respect to the third program. However this is not the case for all experiments, as will be shown in section 2.1.2.

5. The fifth program should perform similar to a Gauss-Seidel implementation running completely in cache, now also executing the Y- and Z-loops. To keep the memory requirements low, only four pairs of neighboring lines in the grid are relaxed. So the Y- and Z-loops run over two different values each, and in total six adjacent lines are relaxed – one pair of lines with red points, on pair below where both red and black points are relaxed, and the bottom-most pair with only black points.

6. To show the effect of the problem getting too large to be kept in cache, the last program performs fused red-black Gauss-Seidel relaxations over a 129^3 grid. The resulting performance decrease, that can be seen as due to this last modification of the test program, is the motivation for cache optimizations in 3D. This size was chosen to be an order of magnitude larger than the L3 caches of all machines under consideration.

First runs revealed problems with the better compilers, when allowing automatic loop interchanges. This could lead to the compiler switching the iteration and the X-loop. This results in unrealistic MFLOPS rates, as the data items are not even loaded from the cache, but kept in a register. To prevent these optimizations, an old Fortran work-around was implemented, as shown in Figure 2.1. The programs always read an integer value from the standard input, and adds this value to the z-coordinate in the iteration loop. When a zero is passed to the program this way, it behaves just like it would without this modification, but, as the compiler is luckily not aware of this, it cannot interchange the two loops anymore.

Early runs used the bandwise data layout described in Section 3.1, but no good MFLOPS rates were achievable, as for many different paddings and problem sizes cache-thrashing could not be completely avoided. To obviate this, an equation-oriented data layout was implemented.

2.1.2 Results

General performance

All results of the detailed performance evaluation show the same basic trend, as depicted in Figure 2.2 for different platforms. High MFLOPS rates are obtained with the first three test problems. On Platform C (Compaq XP 1000 machines, see Appendix A for a detailed description of the platforms) there is a performance decrease due to the introduction of the Y- and Z-loops, but still all machines perform well up to program five, which runs over a small part of the grid with fused relaxation. Hence, this basic red-black Gauss-Seidel kernel can perform well, can be optimized by different modern compilers, and runs with high MFLOPS rates. Note that these rates are still significantly below the theoretical limit of these machines. As shown in [Wei01] this can be explained by the structure of the assembler code, more specifically the mixture of integer operations (e.g. load/store operations, index calculations etc.) and floating point operations. This distribution of operations can be used to determine a theoretical upper performance limit. Assume that these two kinds of operations can be executed in parallel, and that there is the same number of floating point

```
integer null
read *,null
do iter=1, ITER
  do x=1,N-1
    RELAX(x,y,z)
  enddo
  z= z+null
enddo
```

Figure 2.1: To prevent the compiler from interchanging the iteration and x-loop an integer with the value zero is added to the z-coordinate in each iteration.

and integer units, as is for example the case on Platform A (a DEC PWS 500au). Now, when there are twice as much integer operations as floating point operations, the machine will only reach a MFLOPS performance of half the maximum CPU speed, due to the larger amount of integer operations that need to be executed. Furthermore, the test programs described here execute completely in cache. However no real multigrid application is likely to achieve this. Thus the performance measured for real multigrid applications will always be below the performance of this test program.

All graphs also show the expected performance decrease when the problem size gets too big to fit in cache. The performance decreases from around 200 MFLOPS for Platform A to below 50 MFLOPS. On Platform F (a Pentium 4) it decreases from 500 to about 90 MFLOPS. This motivates extensive cache-optimizations, although, as mentioned before, these 200 or 500 MFLOPS, respectively, are unlikely to be reached.

Biased measurements

Although the graphs in Figure 2.3 show the same trends as those in Figure 2.2, they are not so easily interpretable, as there are several variations for the different programs that cannot be easily explained. The graph for Platform E (AMD Athlon CPUs) shows the results for compilation with an older GNU Fortran (Version 2.91 instead of 2.95 on Platform D), that fails to optimize the Gauss-Seidel kernel effectively, once the loop structure gets more complex. The other graph shows the performances for a newer DEC Fortran90 compiler (version 5.3) on Platform B. This performance seems to be disappointing compared to the older DEC Fortran77 (Version 5.1 on Platform A), and the extraordinarily low performance for the second test program with variable coefficients seems to be a consequence of very poor software pipeline optimization.

2.2 Function distribution

To focus the optimizations on the most computationally expensive part of the multigrid program, the work distribution was measured on Platform A using DCPI. For a 3D program the relaxation function needs around 80% of the total execution cycles, the restriction uses 15%, and the correction around 5%. The initialization and other parts were not included in the measurement.

Problem size	Relaxation	Restriction	Correction
8^3	81.56%	14.29%	4.15%
16^3	80.94%	14.75%	4.31%
32^3	78.83%	15.70%	5.47%
64^3	82.36%	14.40%	3.24%
128^3	79.74%	14.83%	5.43%

These numbers justify the concentration of optimization efforts on the relaxation method. The restriction could also be merged into the relaxation due to the similar access pattern (see [Wei01] for details), but to maintain the modularity of the multigrid program, this technique was not implemented within the scope of this thesis. For specific problems an optimized version of the program could be written, but for general problems the possibility to easily replace a given restriction or correction method is necessary.

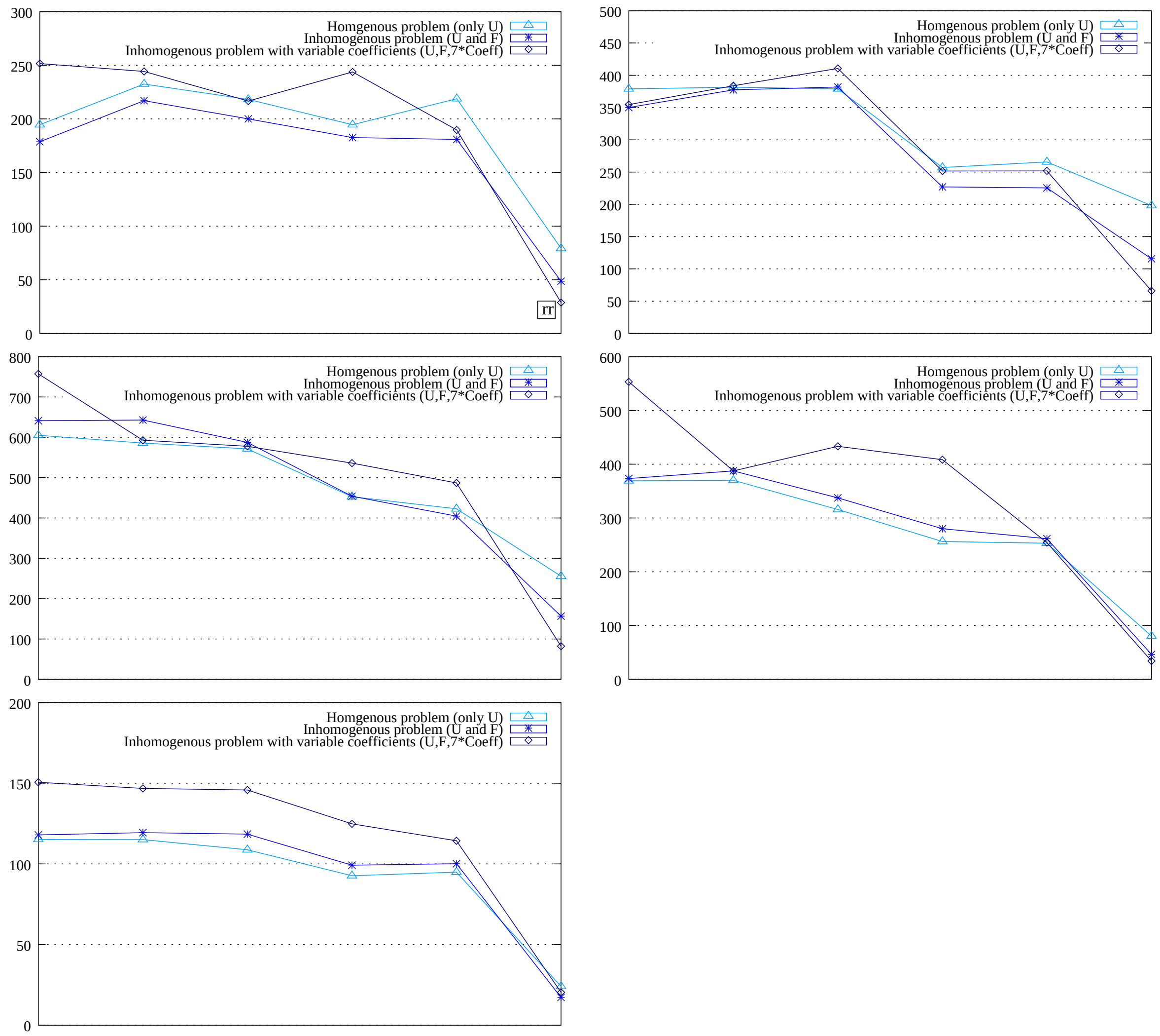

Figure 2.2: Performance results on Platform A (top left), Platform C (top right), Platform F (middle left), Platform D (middle right) and Platform G (bottom left). The small *rr* indicates a run that can be seen in other pictures, for a similar configuration (standard relaxation, 129^3 grid), too.

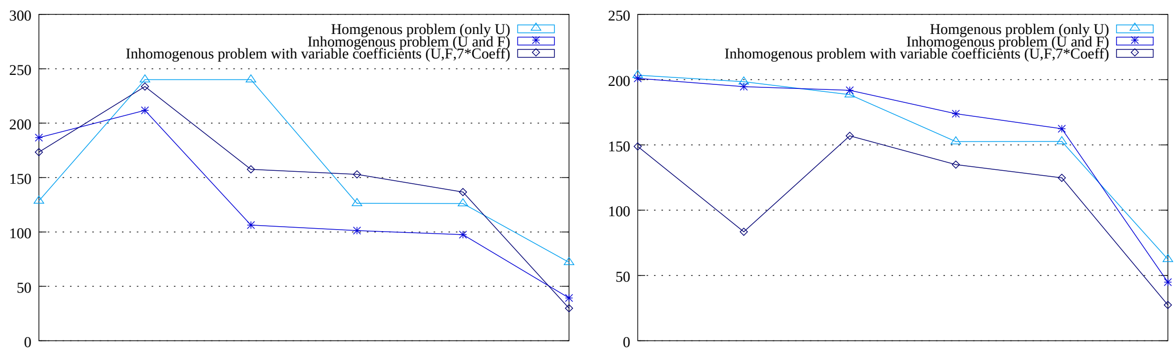

Figure 2.3: Performance results for Platform B on the left, and Platform E on the right.

Chapter 3

Data Layout Optimizations

Three typical data layouts will be discussed in this chapter, including a efficient Fortran implementation for the access-oriented data layout, which yields the best performance results.

3.1 Data Layouts

Analogous to the 2D case, there are many different approaches to the memory layout of the problem data. Read [Wei01] and [KWR02] for a detailed description and measurements for 2D problems. Only three typical approaches will be discussed in this thesis, namely, the access-oriented, the bandwise, and the equation oriented data layouts.

The first solution that comes to mind is, to separately store the unknowns u, the right hand side f, and all coefficients (n, e, s, w, t, b, c for north, east, south, west, top, bottom and center, respectively) in arrays (see Figure 3.1). This is called a bandwise data layout, and may be easy to implement, but is not good in terms of cache performance. The cache line usage, and thus the exploitation of spatial locality of the algorithm is good, but it is very difficult to avoid cache thrashing. The access patterns for u and the other arrays are different, which makes it hard to avoid the eviction of one data element upon access of another, and our experiments have shown, that it is very hard to find a reasonable padding even for small problem sizes.

Another approach is to separate the two different access structures of a Gauss-Seidel smoother. The unknowns usually are accessed in a region around a certain point defined by the stencil, while the right hand side, and the coefficients are only needed for the current point, and not for its neighbors that are used for the new approximation. This separation of the data into two arrays, one for the unknowns, and one for the right hand side together with the coefficients for each point, is called access-oriented data layout. The second array holds the data in blocks for each point, as shown in the middle of Figure 3.1. This data layout also exploits spatial locality very well, as for an access to an unknown, its neighbors in the same cache line are also stored in cache. Moreover, upon

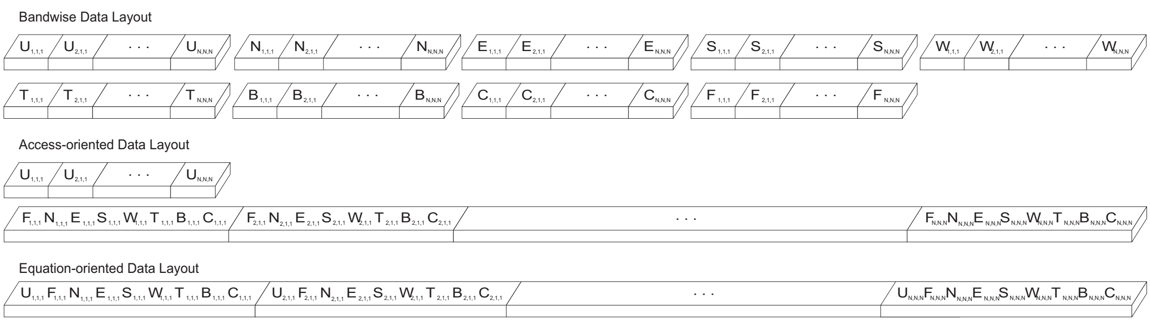

Figure 3.1: Memory organization for the three different data layouts. At the top the nine separate arrays for the bandwise data layout can be seen. Below the array for the unknowns, and the array for the remaining variables are depicted (access-oriented data layout). The equation-oriented data layout uses only a single array for all data.

access of a right hand side value, the coefficients for the corresponding unknown are also needed, and due to the grouped data layout, these accesses can be satisfied from the same cache line.

If the data is stored according to a single equation of the problem, meaning that each unknown is grouped with its right hand side, together with coefficients needed to solve this unknown's equation (Figure 3.1 bottom), this is, not surprisingly, called an equation-oriented data layout. For this layout, upon access of an unknown, it's coefficients and right hand side are also loaded into the cache. While this is certainly a positive effect, the drawback in comparison to the equation oriented data layout is, that for all neighboring unknowns these values are also loaded, although normally not needed. As the reuse of right hand side and coefficients is only across different Gauss-Seidel iterations, this results in many data items that are unnecessarily loaded and stored in cache. On the other hand, for blocked accesses (see Section 4.6 and 4.7), an equation oriented data layout can reduce the number of TLB misses, as the needed data is more likely to be stored in the same memory page. In experiments, this leads to no visible performance improvements, due to the overall bad data cache utilization.

A performance comparison of the data layouts can be seen in Figure 3.2. Although the equation-oriented layout performs very well for small problem sizes, as it minimizes conflict misses and can be easily padded, the performance decreases drastically for bigger problems. On Platform F, the performance for a 129^3 grid is roughly half that of the other two layouts. The bandwise data layout performs better for these larger problems, but even for small problems induces many conflict misses, and is very difficult to pad properly. The best performance can be seen for the access-oriented data layout. On Platform A it is around 30 per cent faster than the bandwise storage, while on other machines this gain is not so obvious. Still, the performance with access-oriented data layout is the highest for problem sizes of 64^3 and 129^3 on all machines.

3.2 Data Layout Implementation

To use the padding techniques described in Chapter 6 in a Fortran77 program, it cannot be avoided to reserve a huge workspace array during startup, and then manually allocate space for the needed arrays. Since all functions have to have access to this array, it is declared as a common block. For the implementation of the relaxation, macros were used to separate the implementation of the data layout from the data access.

These macros can be implemented by directly accessing the workspace array, calculating the offsets into it, e.g. `workspace( offset + (Z*(Xdim*Ydim) + Y*(Xdim) + X) )` , where `Xdim`, `Ydim`, and `Zdim` are the dimensions in x,y and z direction, respectively. Another method is to pass the offset to a function, which then treats the pointer that is internally passed to the function as a pointer to a three dimensional array, example code can be seen in Figure 3.3.

Our experiments have shown, that good compilers (like DEC Fortran 7.1 on Platform A) can produce equally fast code with both versions, while other compilers have benefitted from the second version. If this kind of parameter passing is used, the compiler is more easily aware of the structure of the arrays, and the accesses to it, making it easier to perform optimizations.

Furthermore, the second version still allows working with common blocks and no extra parameters for all other methods, that are not performance critical. The offset in the common block can then only be passed, as described above, to functions like smoothing or restriction , while the remaining ones can still work with the macros and direct workspace accesses, that may be easier to implement.

A traditional padding can be nicely implemented, as all that is necessary is a modification of the `grid` array, e.g. `double precision grid(0:Xdim + XPAD,0:Ydim + YPAD,0:Zdim)`. The non-standard padding is not directly supported by this kind of parameter declaration. The X-padding can still be implemented in the array declaration, but to use the non-standard Y-padding, the access to the array has to be changed. Access of a grid point `grid(x,y,z)` has to be changed to `grid(x+(z*YPAD),y,z)`. This actually violates the Fortran array declaration, but if the array is initialized in a workspace according to the described non-standard padding, this will yield the expected results. As can be seen in the code fragment above, the offset of each XY-plane in the grid is added to the X-coordinate according to the Z-coordinate of the current point. This will make Fortran shift the array access `(z*YPAD)` array elements to the correct element. To prevent calculations upon each access, it is possible to only calculate the offset once, and modify the start

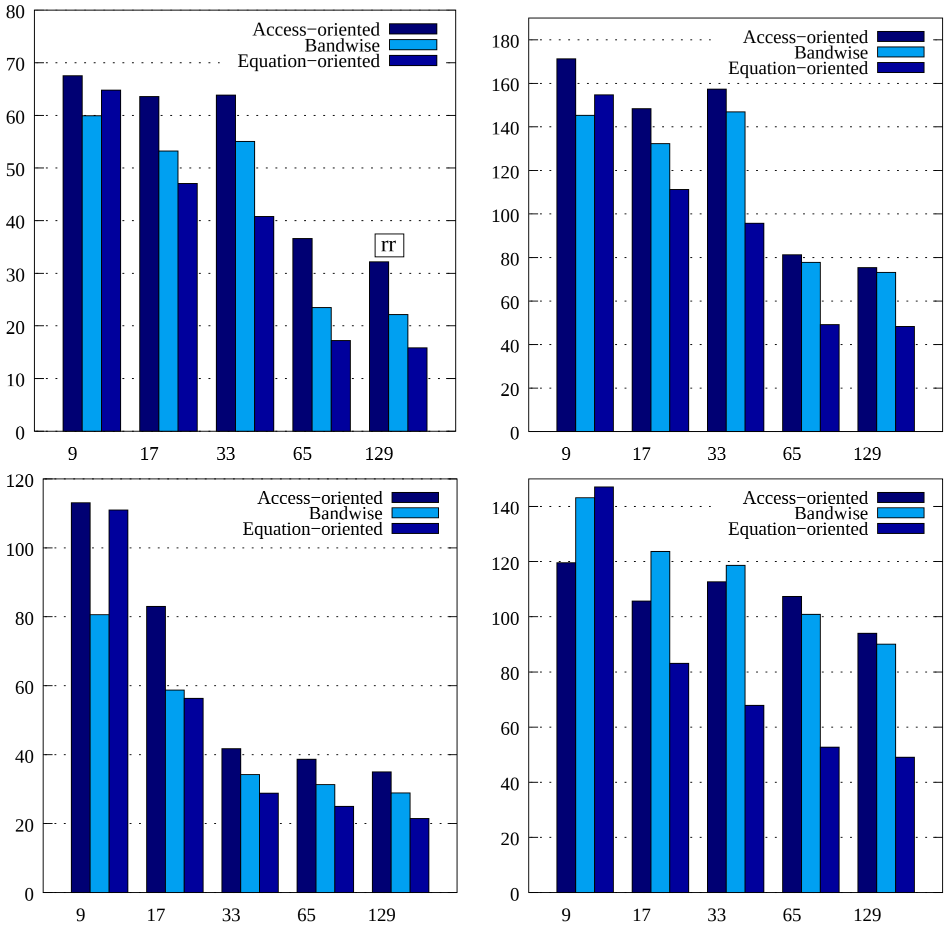

Figure 3.2: MFLOPS rates of the multigrid program, comparing the three different data-layouts for different problem sizes. All MFLOPS rates were measured using a suitable padding for each problem size and data layout, and the best possible loop order. Top left for Platform A, top right Platform C, bottom left Platform D, and Platform F bottom right.

```
program multigrid
  ... [init arrays, dimensions and offsets]
  call rbgs( workspace( offset ), Xdim,Ydim,Zdim)
  ... [process results]
end

subroutine rbgs( grid, Xdim,Ydim,Zdim )
integer Xdim,Ydim,Zdim
double precision grid(0:Xdim,0:Ydim,0:Zdim)

... [perform gauss seidel iteration]

end
```

Figure 3.3: Example of passing a workspace offset to a function, that treats it as a 3D array.

and end points for loops running over an region with constant z-coordinate.

3.3 Architectural Dependencies

The effect of padding differs greatly when using different data layouts and architectures. MFLOPS rates for the three data layouts presented in Section 3.1 on different architectures with various padding sizes can be seen in Figure 3.4.

It is obvious that on all architectures, the low MFLOPS rates of the equation-oriented data layout cannot be increased much by padding. Although the bandwise data layout is faster on Platform C when no padding is applied, the access-oriented data layout is faster on all architectures under consideration when a suitable padding is found.

The different effects of padding can be seen when only the access-oriented data layout is compared across the platforms. While on Platform A and D only a few MFLOPS peaks are visible, Platform C shows more of these. However, on Platform F the program is nearly as fast without padding (the left-most point), as it is with suitable paddings. Some paddings only lead to a significant performance decrease, and although the effect is not always positive, this Platform (with an Intel Pentium 4 CPU) is affected most by the varying padding sizes.

3.4 Loop order

Due to the column-major storage format in Fortran, it is important, that the innermost loop runs over the first index, the next loop over the second index and so on. In this case, the loop kernel accesses array elements in the order in which they are stored in memory, which allows accesses to data items stored in the same cache line without additional main memory accesses. Thus, this loop order is the best to exploit spatial locality.

The effect of a cache aware loop order can be seen in Figure 3.5. The arrays were declared as `double precision u(0:DimX,0:DimY,0:DimZ)` , and so a Gauss-Seidel method with three loops running over X,Y and Z is fastest, if the first loop determines the Z-coordinate, the second determines the Y-, and the third and innermost loop determines the X-coordinate, as this coincides with the storage in memory. The more the loop order differs from this order, the slower the program gets, the MFLOPS rates drop from above 20 to below 15. Once the innermost loop runs over a different direction than X, the level 1 and 2 cache misses increase, and for an innermost Z-loop there is a high number of level 2 and 3 cache misses, due to the lessened spatial locality. Furthermore, the number of TLB misses greatly increases for innermost Y-loops, and is again doubled for innermost Z-loops. Note that this does for example not mean, that the performance of the method is halved, as the performance is generally not only determined by TLB misses, or cache misses as a whole, but also by, for example, pipeline effects and register dependencies. So once the TLB misses are reduced to nearly zero, as is the case with innermost X-loops, other effects like level 3 cache misses lessen this positive impact.

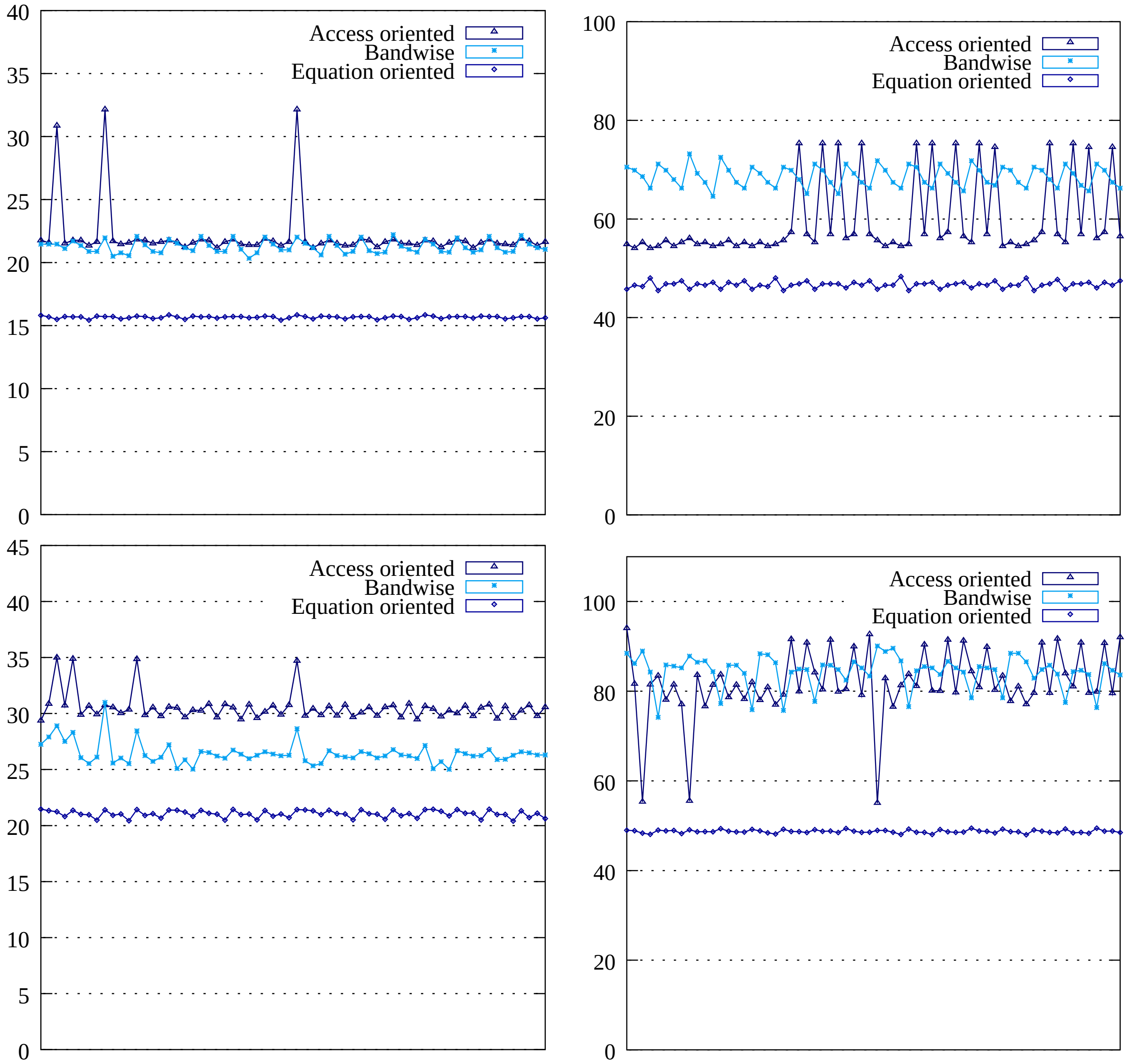

Figure 3.4: MFLOPS rates for different paddings on various architectures, using the whole multigrid program, a problem size of 129^3 and the three different data layouts. Platform A top left, Platform C top right, Platform D bottom left and Platform F bottom right.

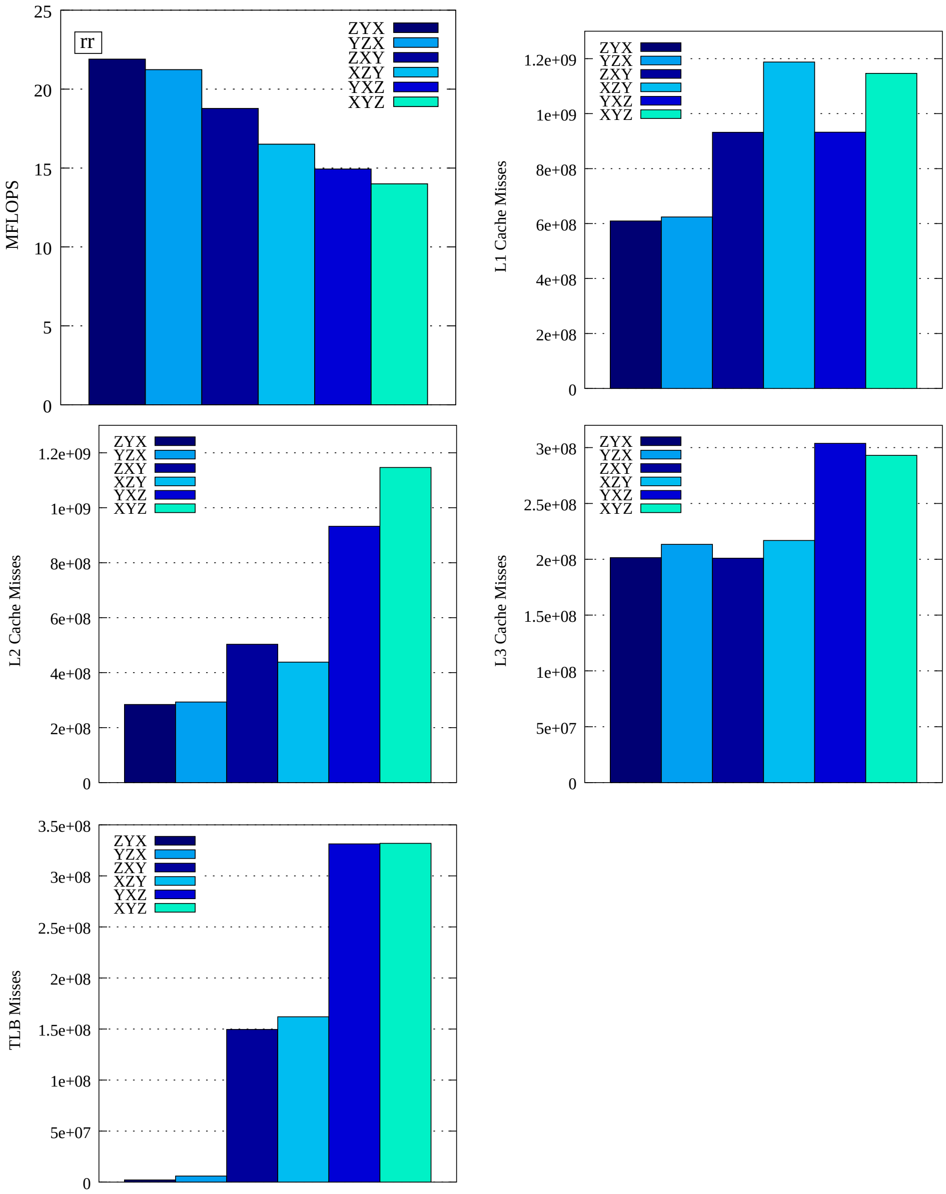

Figure 3.5: Performance results on Platform A for a standard red-black Gauss-Seidel method with different loop interchanges, on a 129^3 grid, using an access-oriented data layout. A reasonable padding was used for all programs, in contrast to other measurements, where the best padding was determined separately for each program. The top left graph shows the MFLOPS rates, top right the L1 cache misses, middle left L2 cache misses, middle right L3 cache misses, and bottom left the TLB misses. The bars were ordered in descending MFLOPS order (see top left graph).

Chapter 4

Data Access Optimizations

As shown in Section 2.2 the smoothing component of a multigrid solver is the most time-consuming part, and it has been shown in 2D [WKKR99], that cache optimizations can speed up the computation considerably. To improve the data locality, and avoid main memory accesses, the traversal-order of the Gauss-Seidel method can be changed to improve the possibility of memory accesses to be satisfied by a higher level of the memory-hierarchy.

As part of this thesis, six blocking variants are explained and benchmarked. Furthermore, a Red-Black Gauss-Seidel will be assumed from now on, as its parallelization and convergence-rate have been shown to make it very suitable for multigrid calculations. For the following sections, when considering the amount of data to be held in the cache, it is assumed that this data is cached in a non-conflicting way. The problem of cache-thrashing and the determination of non-conflicting tile-sizes is discussed in more detail in Chapter 6.

4.1 Notation of Gauss Seidel Variants

To identify the variants selected, it was necessary to choose a naming convention uniquely identifying them. Several approaches are possible: using geometry dimensions, movement dimensions or the number of blocked loops.

4.1.1 Geometry Dimensions

Considering the number of dimensions of the shape moved through the grid, the names *3D blocking* for a small cuboid being moved through the grid, and *2D blocking* for a plane-wise movement are intuitive. The problem is, that there is no real *1D blocking*, as all these would have the same effect, similar to the *2D blocking*, of first finishing the relaxation of one plane, before moving on to the next.

4.1.2 Movement Dimensions

When, instead, choosing names regarding the movement dimension of the shape, *3D blocking* remains the same. A problem is the discontinuity of *2D blocking* and *1D blocking*, as the shape for *2D blocking* would be a line in the cube (a one-dimensional shape), while *1D blocking* requires a plane (two-dimensional) to be moved through the cube. This switching of dimensions (2D blocking with 1D shape, and 1D blocking with 2D shape) makes it hard to intuitively identify a blocking method.

4.1.3 Number of blocked Loops

To identify the method by counting the number of blocked loops is non-ambiguous. In our standard implementation there are four loops: the X,Y,Z and iterations loop, all of which can be blocked, resulting in *1-way blocking* to *4-way blocking*. Note that in this case it is **not** recommended to talk of *n 'D' blocking*, as this would allow *4D blocking* to be performed on a 3D problem, which sounds very un-intuitive. Similar to the other two terminologies, the problem of loop interchanges remains, so for unique identification of a blocking implementation, the loop order also has to be considered. In this chapter this last naming convention will be used, and the 1-way loop interchanged variants will all be discussed in Section 4.4.

4.2 Standard implementations

The standard implementation of a red-black Gauss-Seidel method, based on the discretization and stencil described in Section 1.1, is shown in Figure 4.1.

This straightforward implementation already has the most efficient ordering of loops, given a standard Fortran-like declaration of the arrays with column-major data layout. As shown in Figure 3.5, the incorrect loop order can lead to very low MFLOPS rates. This is due to a resulting stride not equal to one, but to N or in the worst case N^2. The cache-lines that are loaded due to access to a single memory location `(x,y,z)` also contain data items with neighboring X-values, so the innermost loop should naturally run over these values.

The two separate relaxations of red and black points require a total of 2ν sweeps over the grid (where ν is the number of iterations that needs to be performed), and the whole data of the grid to fit into cache. If this is not the case, the red points needed will be evicted from cache, and the black points requiring them will lead to a second main memory access. Let U be the size of a single point in the grid, and V the size of the data necessary to relax it (in our example implementation the right-hand side, and 7 coefficients, thus 64 bytes), then this requires the cache to hold $N^3 * U + (N-2)^3 * V$ bytes in order that this code can be executed efficiently ($N = 2^k + 1$ is the size of the arrays in x,y and z direction).

4.3 Fused

The first real optimization is to avoid the two complete traversals of the grid, and fuse the red and black loops into one. As shown in Figure 4.3a this leads to a relaxation of a red point in plane i and directly afterwards to that of the black point below in plane $i-1$. The black neighboring point can be updated without violating any data-dependencies, as all 6 surrounding red points are now updated correctly.

The four red points surrounding this black point were relaxed during the last iteration of the Z-loop, and the red point below two iterations ago. This only requires ν sweeps over the grid, instead of 2ν in the standard case.The code for this fused main loop can be seen in Figure 4.2, the resulting access pattern is illustrated in Figure 4.3a. The fusing of the loops, analogously to 2D requires special treatment of the planes with $Z = 1$ and $Z = N-1$, as these cannot be updated with the described loop kernel. Before the loop from Figure 4.2 starts, the red points of the first plane, $Z = 1$, already have to be updated once, while the black points of the last plane, $Z = N-1$, have to be updated as a final step. This kind of preparation is common for all subsequent Gauss-Seidel variants. The more irregular the shape of the domain for updating the points in the innermost loop gets, the more planes on the sides of the grid have to be handled separately.

This fusion requires a total of 4 planes to fit in cache. After the black point is updated, the red one below is never needed again for this iteration. This still requires $N^2 * 4 * U + (N-2)^2 * 2 * V$ bytes, which will quickly exceed modern cache sizes for finer grids.

4.4 1-Way Blocking

While loop-fusion combines the red- and black-point sweep, the next logical step is to block the iteration loop. This by itself only introduces another loop without changing the update order, but if the new loop if moved into the X-loop, multiple red-black pairs of unknowns will be updated in one step. The update order then is similar to that of the fused method, and is shown in Figure 4.3b. An exemplary implementation can be seen in Figure 4.4. Assuming that $N^2 * (2 * B_K + 2) * U + (N-2)^2 * (2 * B_K) * V$ floating point numbers can be held in cache, these values will be reused as often as necessary. Assuming that all iterations were blocked, after the last relaxation they will not be relaxed again during the execution of the method. It is only necessary to traverse the whole grid ν / B_K times in this case.

```
C perform NU iterations
do k=1,NU
  C relax red points
  do z=1,N-1
    do y=1,N-1
      do x=2-mod(y,z,2), N-1, 2
        RELAX(x,y,z)
      enddo
    enddo
  enddo
  C relax black points
  do z=1,N-1
    do y=1,N-1
      do x=1+mod(y,z,2), N-1, 2
        RELAX(x,y,z)
      enddo
    enddo
  enddo
enddo
```

Figure 4.1: Pseudo-code for standard red-black Gauss-Seidel implementation, the RELAX macro calculates the following: $U_{x,y,z} = (F_{x,y,z} - W_{x,y,z}U_{x-1,y,z} - E_{x,y,z}U_{x+1,y,z} - S_{x,y,z}U_{x,y-1,z} - N_{x,y,z}U_{x,y+1,z} - B_{x,y,z}U_{x,y,z-1} - T_{x,y,z}U_{x,y,z+1})/C_{x,y,z}$.

```
do k=1,NU
  do z=1,N-1
    do y=1,N-1
      do x=2-mod(y,z,2), N-1, 2
        C relax red point and black point below
        RELAX(x,y,z)
        RELAX(x,y,z-1)
      enddo
    enddo
  enddo
enddo
```

Figure 4.2: Pseudo-code for fused red-black Gauss-Seidel implementation.

A different relaxation-order is achieved by interchanging the k- and the X/Y-loops. In this case, all red points in the Z-plane are relaxed, continuing with the black points in plane $Z-1$ and succeeding planes, as determined by the iteration blocking factor. Another variation is, of course, to only interchange k- and x-loop. The method then traverses the grid in a line-wise manner. Still, for these two variants the number of points to be kept in cache remains the same, but differences may be seen due to the different behavior of the innermost loop. In most cases, the x-loop will be repeated more often than the k-loop, allowing the compiler to perform more pipeline-optimizations, and reduce the number of branch operations and mispredictions. The differences of the loop interchanges for 1-way blocking are illustrated in Figure 4.5.

In the experiments performed, another version of this method was introduced. For the standard case of a V(2,2)-cycle, a hand-unrolled version of the code in Figure 4.4 was used. Theoretically this code should perform similar to the normal version, as the blocking-constants were known to the compiler. Practically, depending on the compiler used, noticeable differences in performance were observed.

4.5 2-Way Blocking

As for bigger problem sizes the required number of planes will not fit in cache, it is necessary to reduce the working set even more, by blocking the next loop, normally the x-loop. In this case

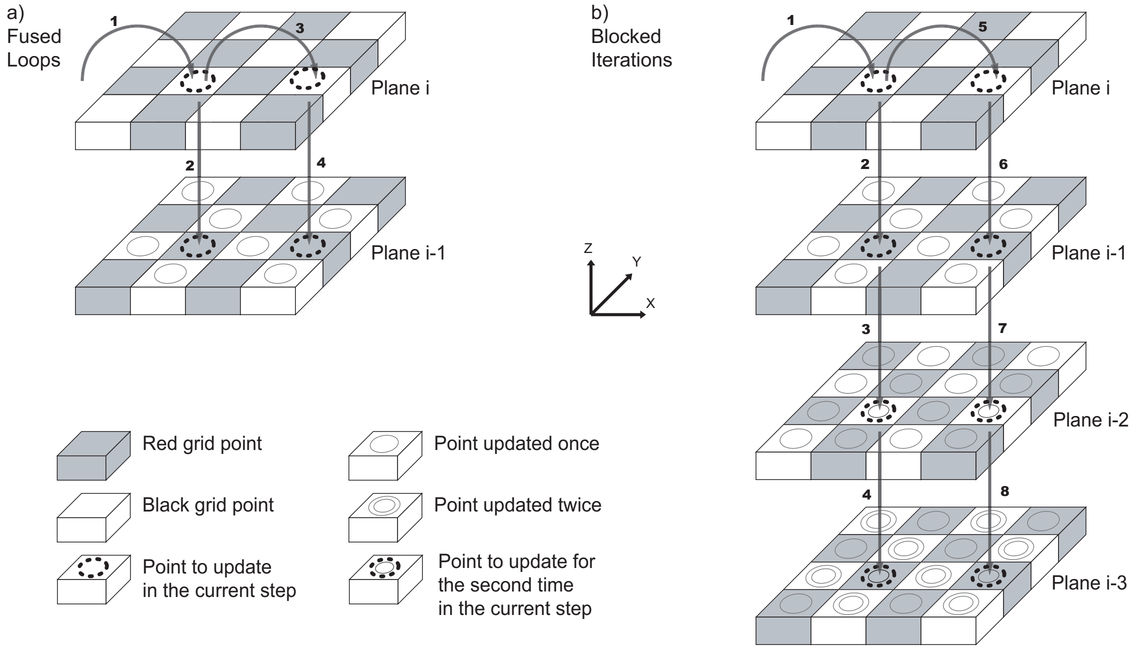

Figure 4.3: a) Fused Loops: the red point in plane *i* and afterwards the black point below in plane *i-1* are relaxed together b) Blocked Iterations: displayed are two blocked iterations, red and black points in different planes are updated in one step.

line-segments of red and black points in adjacent planes are relaxed, with a memory-requirement of about $3 * (B_X + 2) * (2 * B_K + 2) * U + B_X * (2 * B_K) * V$ when blocking the X-loop by B_X.

4.6 3-Way Blocking

The disadvantage of the 2-way blocking technique is the fact that the reuse of the blocked loops is relatively bad, as only points along the blocked line-segment are reused, but none with different Y-values. Hence, the Y-loop can be blocked, to also enable a reuse of neighboring points in the same plane. 3-way blocking can be imagined as partitioning the plane into smaller rectangular areas called *tiles*, first relaxing all red points there, then moving on to the black points of plane $Z - 1$, and so on. When all tiles along the Z-axis are relaxed ($2 * B_K$ in this implementation), the next tile in plane Z is selected, as shown in Figure 4.7.

Taking into account the intended number of blocked iterations, the tile size (B_X, B_Y) can now be selected according to the cache capacity. The relaxation of one set of tiles now uses roughly

```
do kk=1,NU/BLOCKK, BLOCKK
  do z=1+BLOCKK,N-1
    do y=1,N-1
      do x=2-mod(y,z,2), N-1, 2
        C blocked iteration loop
        do k=0, BLOCKK-1
          RELAX(x,y,z-k)
        enddo
      enddo
    enddo
  enddo
enddo
```

Figure 4.4: Pseudo-code for a red-black Gauss-Seidel implementation with blocked iteration loop and loop interchange.

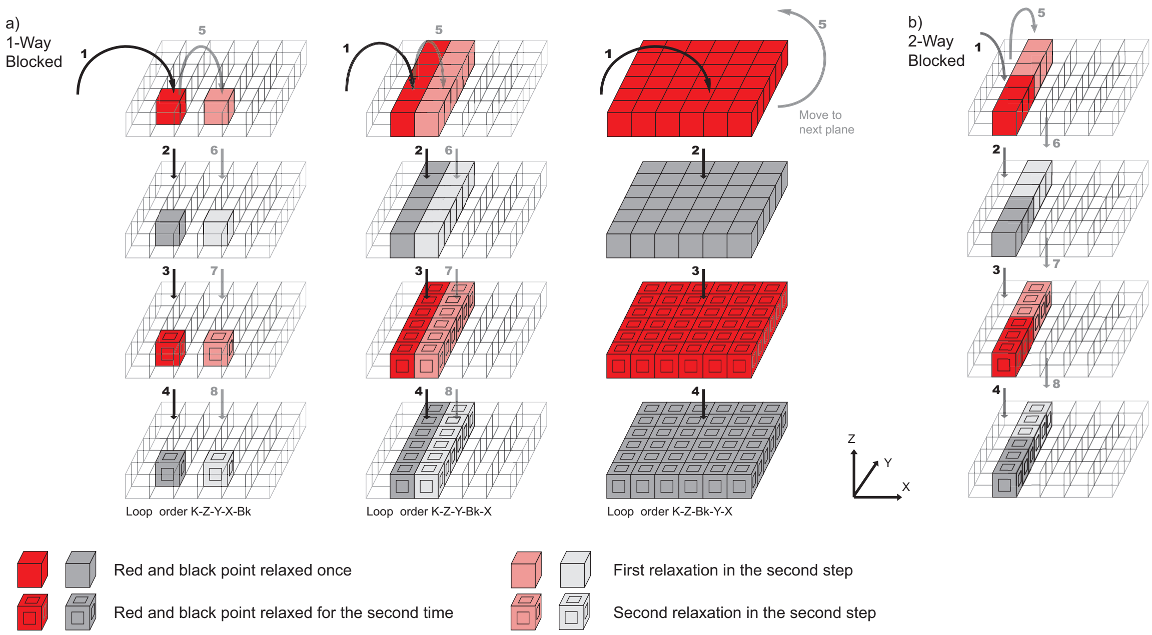

Figure 4.5: a) 1-Way blocking with three different loop interchanges. The loop order is displayed above each set of planes. b) 2-way blocking is achieved by blocking the X-loop of the second 1-way blocked variant with loop order K-Z-Y-Bk-X.

$(B_X + 2) * (B_Y + 2) * (2 * B_K + 2) * U + B_X * B_Y * (2 * B_K) * V$ bytes.

4.7 4-Way Blocking

When all four loops of the standard program from Figure 4.2 are blocked, the program will look as shown in Figure 4.6.

Now, a smaller cuboid with dimensions (B_X, B_Y, B_Z) is moved through the grid. For each blocked iteration the cuboid is shifted by -1 in all three directions to obey all data dependencies. Now the same red points relaxed in the cuboid with $k = 0$ are relaxed for the second time for $k = 2$, likewise the black points for $k = 1$ and $k = 3$, respectively. The shape of these intersecting cuboids is illustrated in Figure 4.7.

The points in the grid are, in contrast to the other methods described, not updated in a plane-wise manner anymore. In previous methods, before moving to the red points in plane $i + 1$, all points in plane i were updated, likewise all black points in plane $i - 1$ were updated, all red ones in plane $i - 2$ for the second time, and so on. Now the updating of the grid points always takes place as soon as possible. Stopping at any point in the relaxation process, there is now a region of untouched points, behind those a layer of red points updated once, then a layer of relaxed black points, then one of red points updated for the second time, until, finally there is a region of points which are updated B_K times (if more than two iterations are blocked there are additional layers).

The drawback for this method is of course the increased overhead, as e.g. all points with $x, y, z < B_K$ have to be prepared accordingly. Furthermore the cache requirements quickly exceed the capacities of most modern architectures, resulting in small values for B_X,B_Y and B_Z. These shortened loops also have the negative effect that the chance of pipeline stalls increases. Thus, it is convenient, when manually choosing the blocking factors, to increase B_X and decrease B_Y and B_Z accordingly. The needed memory can be approximated by $(B_X + 2 + B_K) * (B_Y + 2 + B_K) * (B_Z + 2 + B_K) * U + (B_X + B_K) * (B_Y + B_K) * (B_Z + B_K) * V$. The approximation error for the memory requirement will of course grow for bigger B_K, but will work fine for standard values around two.

```
do kk=1,NU/BLOCKK, BLOCKK
  do zz=1+BLOCKK,N-1, BLOCKZ
    do yy=1+BLOCKK,N-1, BLOCKY
      do xx=1+BLOCKK,N-1, BLOCKX

        C relax points in (BLOCKX,BLOCKY,BLOCKZ) cuboid
        do k=0, BLOCKK-1
          do z=zz, min(N-1,zz+BLOCKZ)
            do y=yy, min(N-1,yy+BLOCKY)
              do x=xx+mod(xx+y+z ,2), min(N-1,xx+BLOCKX), 2
                RELAX(x-k,y-k,z-k)
              enddo
            enddo
          enddo
        enddo

      enddo
    enddo
  enddo
enddo
```

Figure 4.6: Pseudo-code for a red-black Gauss-Seidel implementation with 4-way blocking.

4.8 Skewed Blocking

For 2D, a blocked method using skewed loops is evaluated in [Wei01, Pfä00]. As this technique can not be directly adapted to 3D, due to the more complex data dependencies, loop skewing will not be discussed in this thesis. Problems arise, when, across different planes, the tiles of for example a 4-way blocked Gauss-Seidel implementation are skewed. In cases like this, the alternating structure of red and black points in adjacent planes quickly leads to violations of data dependencies.

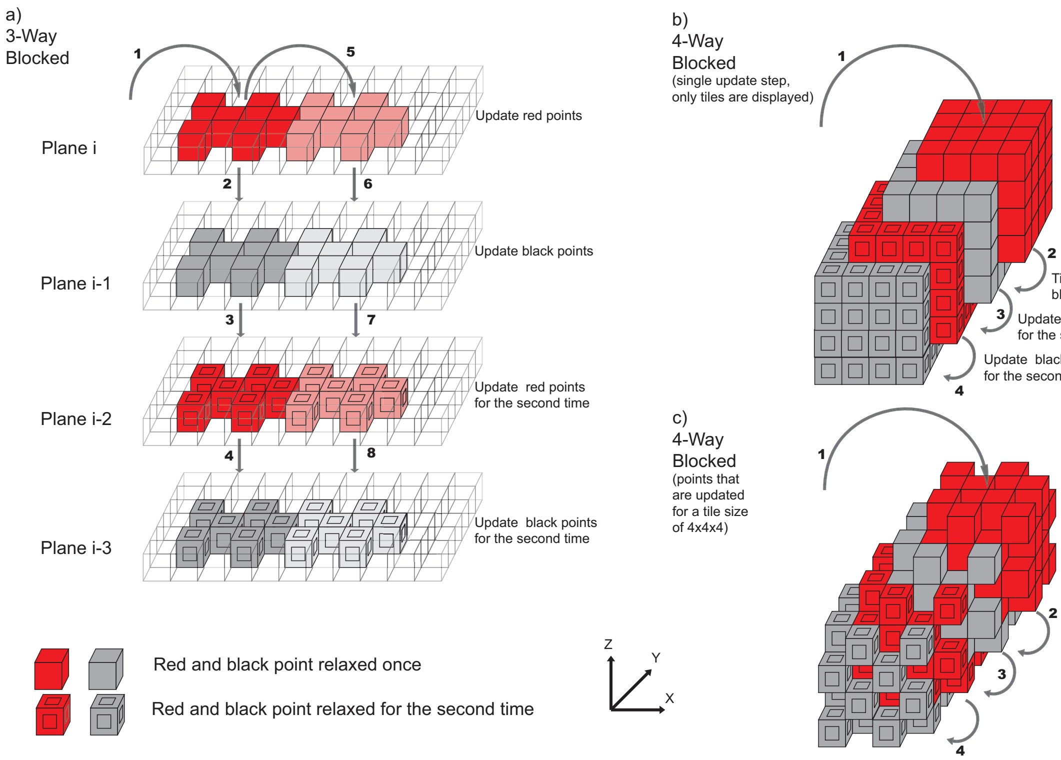

Figure 4.7: a) Update step for 3-way blocked red black Gauss-Seidel b) Tiles that are updated for a 4-way blocked method c) Points that are updated for 4-way blocking and a tile size of 4x4x4.

Chapter 5

Results

The performance results for the data access and layout optimizations will be presented and explained in this chapter. Furthermore, results for homogenous problems, and problems with constant coefficients will be shown.

5.1 Variable coefficients

Red black Gauss-Seidel implementations using the different blocking techniques where tested on different platforms, using problem sizes of 9^3, 17^3, 33^3, 64^3 and 129^3. While the smallest problem only needs 100KB, the largest one uses almost 300MB, which exceeds the level 3 cache size of all machines. For details on the Platforms used please read Appendix A. The graphs show a comparison for the performance for the whole multigrid application using a V(2,2) cycle, instead of only the smoothing function. If the chosen blocking factors get too large for the coarser grids, the standard Gauss-Seidel function with cache-aware loop order is executed.

The first two bars in each graph show the importance of padding, which will be explained in more detail in Chapter 6. For all other runs, suitable padding and tile sizes where determined using a search of the parameter space.

It can be seen, that the overall performance increase for large problems, is around 20% for Platform F (Figure 5.4) up to 60% on Platform C (Figure 5.3) , and 120% on Platform A (Figure 5.1). For smaller grids, the performance increase varies, but is still 40% on Platform A. The Fused method performs particularly well on all machines for smaller problems, as it implies the lowest loop overhead of all methods, a compact loop kernel, and is very effective, if the whole problem can be held in cache.

Generally, the performance of the more complex variants, especially 3- and 4-way blocking, is disappointing. Only on Platform F, a small increase in performance for 4-way blocking in comparison to a fused traversal can be seen. To investigate this behavior more in detail, DCPI was used on Platform A for a problem size of 129^3. The results can be seen in Figure 5.2. While the level 3 cache misses are decreased by a factor of roughly 2, the TLB misses increase by a factor of more than 3. This can be explained by the much more distributed access order of the 4-way blocking. It accesses data items across many different planes, which quickly fills the TLB, which has a capacity of 128 entries on this platform. Due to the high memory requirements of the problem, the blocking sizes have to be adapted to the level 3 cache to be efficient, but this on the other hand leads to relatively large tiles which then cause problems with TLB misses. To avoid these, for example *data copying* could be used [TGJ93].

Another effect is the relatively small size of the grids. Due to the increased memory requirements for the coefficients and the right-hand side, only grids up to 129^3 nodes were considered. Hence, the points in one plane are still relatively close together. Three lines of the unknown array fit into the cache of all platforms under consideration, allowing to reuse all four neighboring points in the same plane. As the performance results have shown in 2D, the more complex blocking techniques are not efficient for smaller problems, but guarantee a relatively constant performance even for large problems. Furthermore, the performance increase due to reuse across different adjacent planes is rather small, and the growing loop overhead of 2-,3- and 4-way blocking can spoil the performance gains.

On Platform A, the performance of the hand-optimized 1-way blocking variant yields interesting results. In theory, it should perform exactly like the normal 1-way blocking, as the only difference is the innermost loop blocking the iterations, which is not controlled by a constant in this case, but unrolled for 2 blocked iterations by hand. This unrolling allows the compiler, DEC Fortran 7.1 in this case, to better optimize software pipelining and prefetching, due to the simpler structure of the kernel. On other machines this increase is less significant. Furthermore, this kind of unrolling can not be easily done for the 2-, 3- and 4-way blocked methods, and is not a cache optimization per se. It only allows the compiler to more efficiently restructure and pipeline the operations, and more easily recognize the reuse of data items. When too much unrolling is done, the code may not fit in the instruction cache any more and lead to poor performance due to instruction cache misses. This can quickly be the case when effective tile sizes for e.g. 3-way blocking are unrolled.

5.2 Constant coefficients

With constant coefficients, the multigrid program requires two arrays for each grid level; one for the unknowns, and another one for the right-hand side values. In Figure 5.5 it can be seen, that reducing the number of arrays from nine to two, results in a performance gain of roughly 20 MFLOPS for each program on Platform A. The MFLOPS rates for a problem size of 17^3 are even higher than those for 9^3, which is a result of the very short loops and the resulting poor optimizations in the latter case.

For arrays of 129^3 the relative performance of the nine methods is comparable to the variable coefficient case. The 1-way blocked hand optimized version is still the fastest one, and the more complex methods run slower than this and the loop interchanged 1-way blocking. The performance increases by more than 50% for 129^3, and less for the smaller problems.

Again, the 129^3 case was analyzed with DCPI on Platform A, the resulting cache miss rates can be seen in Figure 5.6. While the effects on the Level 1 and 2 Cache Misses vary, the Level 3 Cache Misses are more than halved once blocking is used. In contrast to the variable coefficient case, the TLB Misses now only increase notably for the 4-way blocked method, but it also has the lowest Level 2 Cache miss rate in this case. The effect of the hand unrolled loop for the 1-way blocked method is also not as big, which indicates that the smaller loop kernel, due to the missing coefficients, is better optimized in all cases.

5.3 Constant coefficient and homogenous problems

For the solution of a homogenous problem with constant coefficients, only a single array of unknowns is necessary on the finest grid. Still it is necessary for all coarser grids to include a right hande side due to the structure of the multigrid algorithm. For these grids, a standard relaxation function as described in section 4.2 was used. Hence, the results are not directly comparable to those of section 5.2, as the optimized version of the relaxation function is only used on the finest grid. In this case the 1-way blocked hand optimized version runs around 18% faster than the standard version with padding. The methods employing more blocked loops are considerably faster than the standard method, but still show a disappointing performance. The performance rates for all problem sizes can be seen in Figure 5.7. As the 65^3 problem still fits completely into the Level 3 Cache of Platform A, the performance is much higher than that of the larger problem. For the smallest two problems, the MFLOPS rates are almost identical to the constant coefficient case, and only from 33^3 on, a positive effect of the reduced memory requirement can be seen.

The DCPI results, shown in Figure 5.8, are similar to those of the constant coefficient case. The TLB influence is even smaller for the 4-way blocked method, but the Level 3 Cache Misses are less than half of those of the standard case.

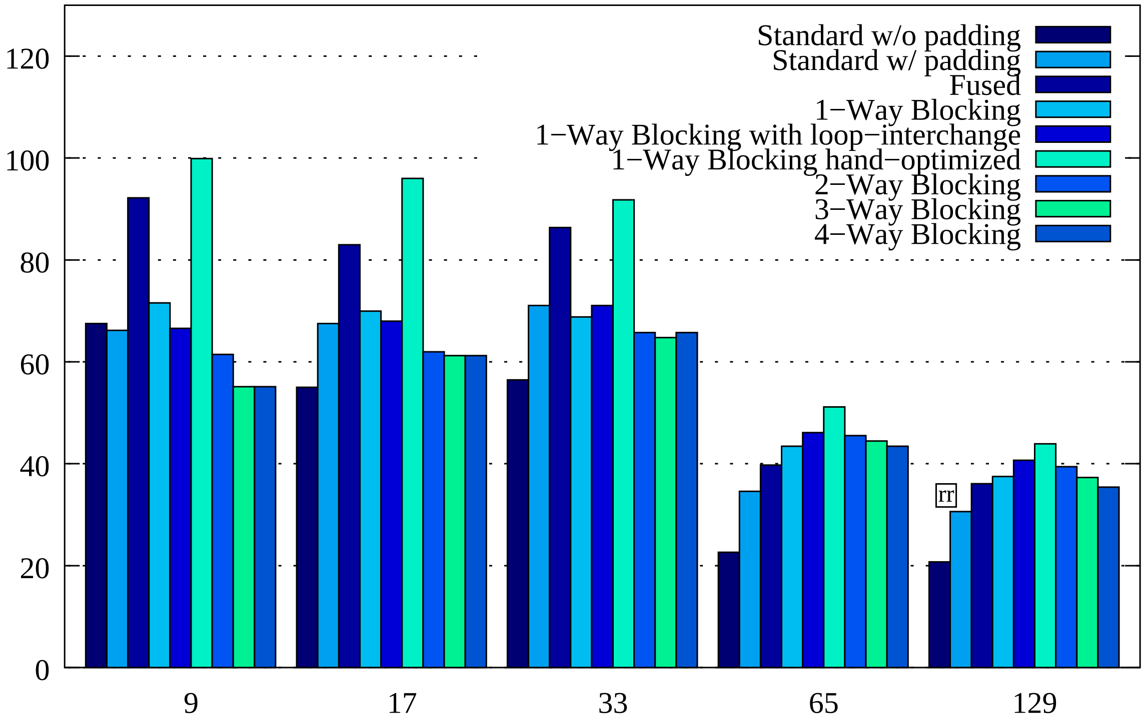

Figure 5.1: MFLOPS rates for multigrid program with problem sizes of 9^3,17^3,33^3, 65^3 and 129^3 using all nine red-black Gauss-Seidel implementations. Measurements were made on Platform A. The *rr* again denotes a standard run with grid size 129^3 and standard relaxation.

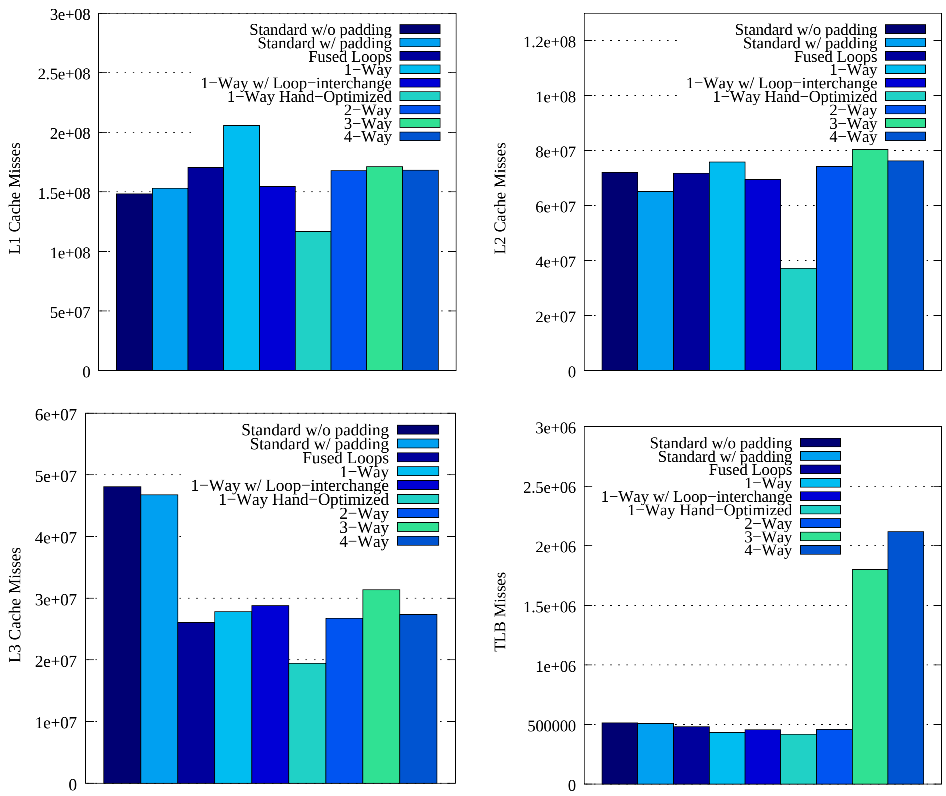

Figure 5.2: Number of L1 Cache Misses (top left), L2 Cache Misses (top right), L3 Cache Misses (bottom left) and TLB Misses (bottom right). All results were measured on Platform A with DCPI for a problem size of 129^3, using the same paddings as in Figure 5.1.

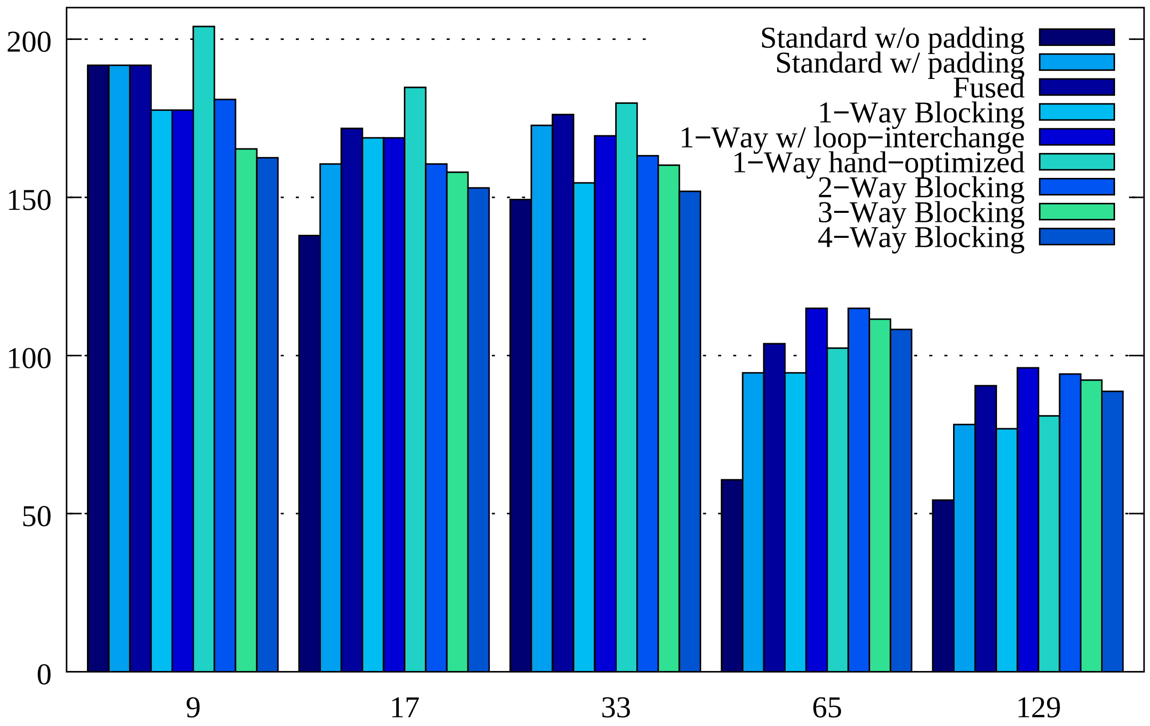

Figure 5.3: MFLOPS rates for multigrid program with problem sizes of 9^3,17^3,33^3, 65^3 and 129^3 using all nine red-black Gauss-Seidel implementations. Measurements were made on Platform C.

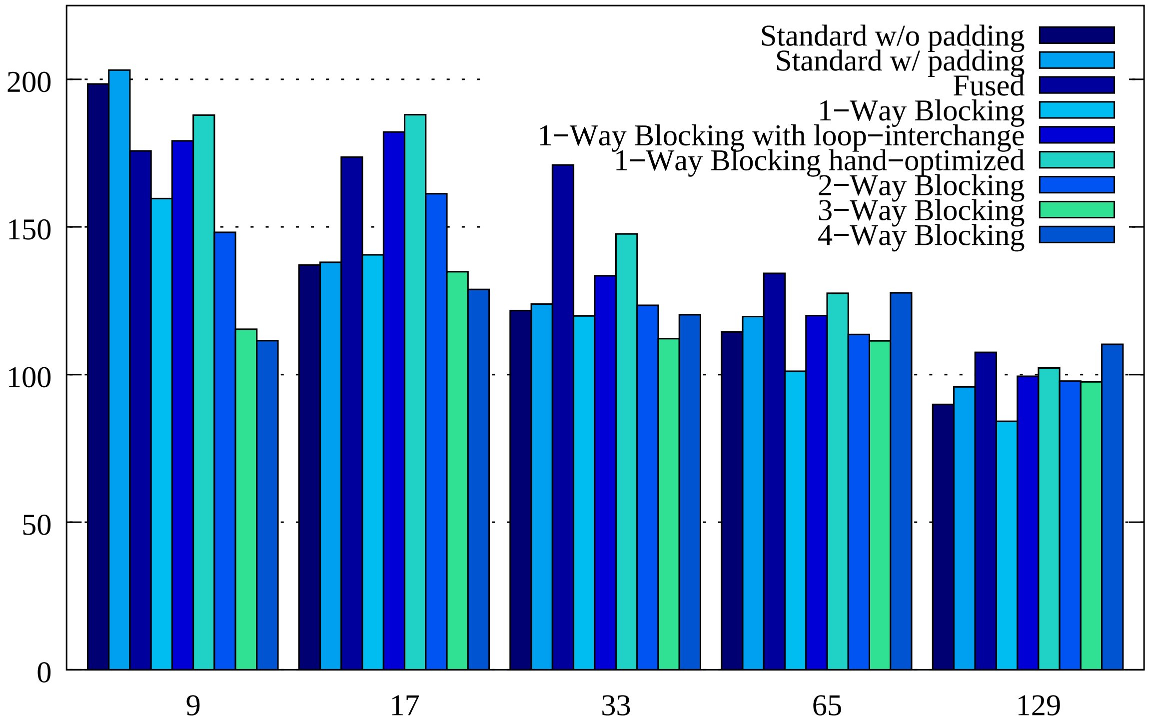

Figure 5.4: MFLOPS rates for multigrid program with problem sizes of 9^3,17^3,33^3, 65^3 and 129^3 using all nine red-black Gauss-Seidel implementations. Measurements were made on Platform F.

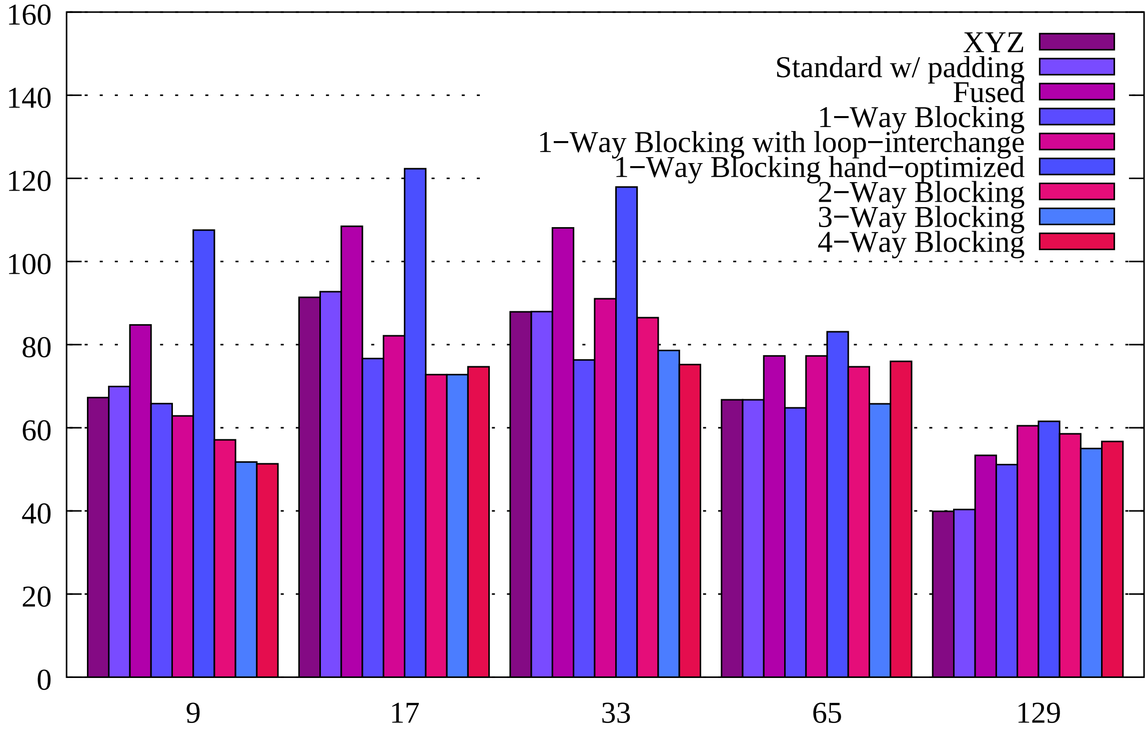

Figure 5.5: MFLOPS for the constant coefficient multigrid program running on Platform A with suitable padding and an inhomogeneous problem using access oriented data layout.

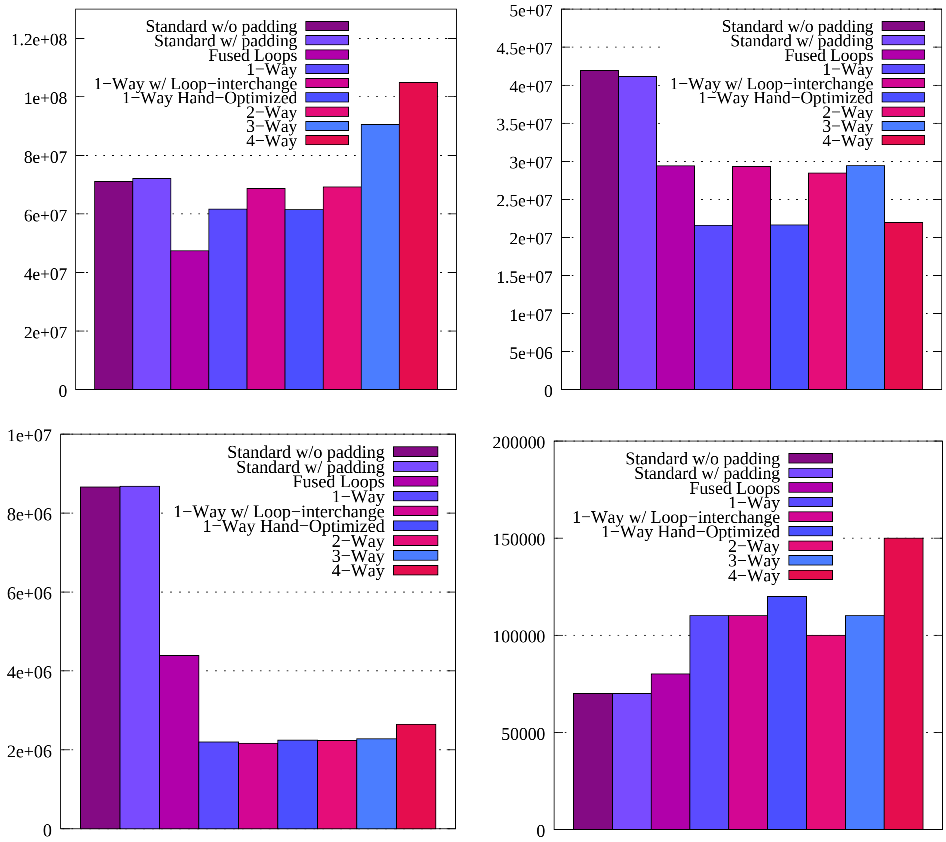

Figure 5.6: DCPI results for the constant coefficient multigrid program with unknowns and right-hand side arrays for all grids.

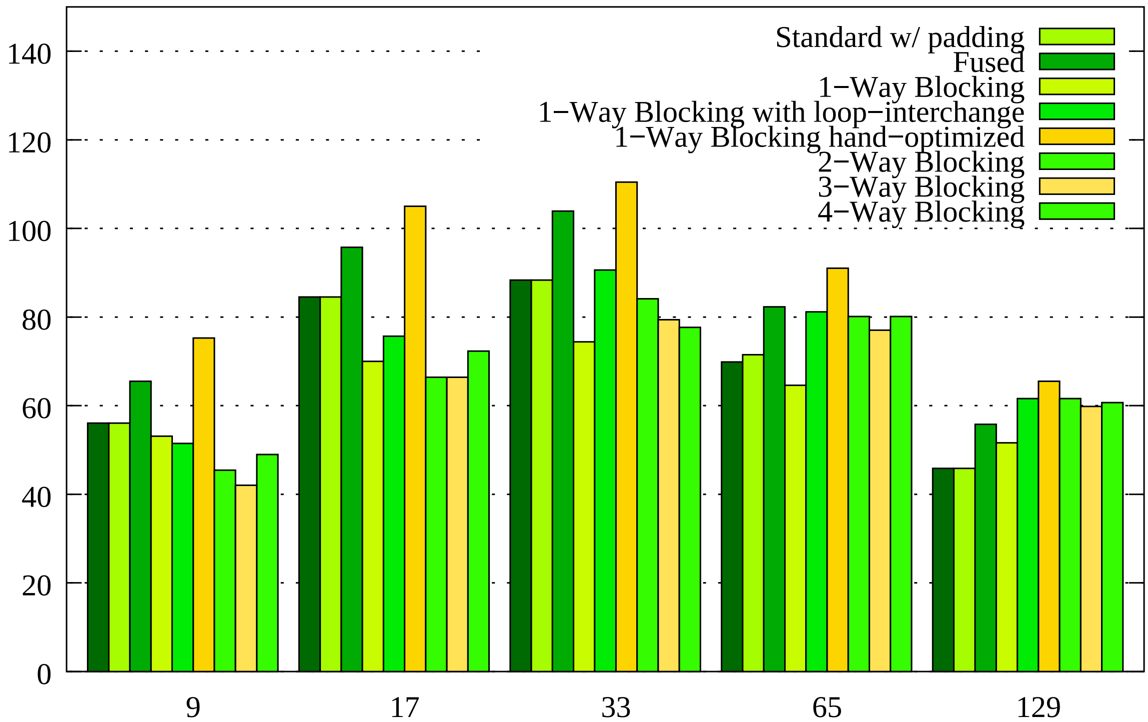

Figure 5.7: MFLOPS for the multigrid program running on Platform A with suitable padding and a homogenous problem with constant coefficients, hence, with only a single array for the unknowns on the finest grid.

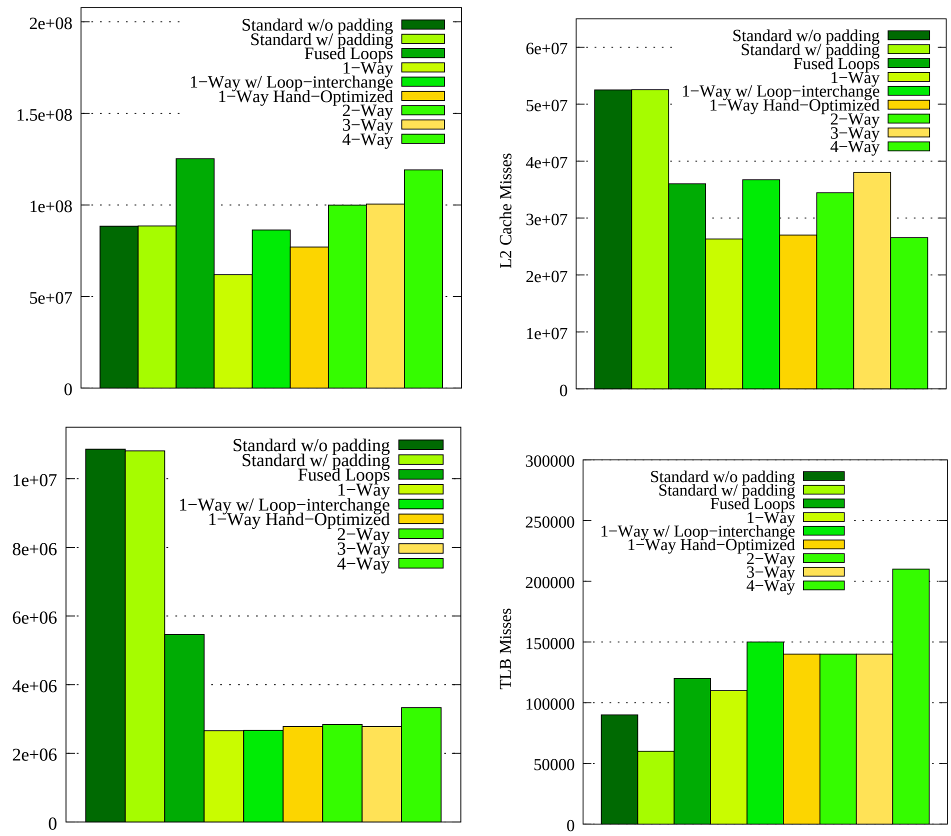

Figure 5.8: DCPI results for the multigrid program only with the array of unknowns on the finest grid.

Chapter 6

Padding

The importance of the padding technique, which was described in Section 1.3, can be seen in the Figures 5.1, 5.3 and 5.4 from Section 5.1. The first two bars for each problem size show the performance increase due to appropriate padding. For a problem size of 129^3 the MFLOPS increase by roughly 6% to 50%.

The difficulty here lies in the selection of the correct parameters, as for the case of an access-oriented data layout, there are 5 parameters for padding alone. Padding in X and Y dimension for each of the two arrays, and an inter-array padding to align the relative addresses for the unknown array and the array holding the coefficients with the right hand side. For the blocked variants of the Gauss-Seidel smoother, the blocking factors also have to be chosen according to the cache size, adding another one to four parameters. The value for blocking the iterations may be fixed due to the V-cycle parameters, which are both 2 in the reference implementation.

Generally, different approaches to the parameter selection can be used. The first, and most direct one, which was also used in Chapter 4 is to exhaustively search the parameter space for the best parameters. This is legitimate when taking into account the speed of hardware and compiler development, and that analytical models can only insufficiently include all factors that may be necessary. As proposed in [WD98], the exhaustive search could be done once upon installation of a software package or library, which could from then on work with values that have empirically shown to be optimal in the parameter space searched.

On the other hand, heuristical and analytical approaches have been proposed by [Riv01, RT00]. In the following, the applicability of one of the analytical approaches described in [Riv01] will be evaluated.

6.1 Padding Size Selection Theory

Once an access to a single data item in a tile leads to a conflict miss, the following accesses will also conflict with other data items in the cache. To select a tile size and suitable padding for the biggest possible tile fitting in a cache with size C, the following formulas have been proposed in [Riv01]. The dimensions of the array are denoted D_x,D_y, and D_z, respectively, while T_z determines the minimum number of planes which have to be held in cache. T_z is determined by the way the method is blocked. For a fused Gauss-Seidel, T_z equals four, while e.g. for 1-way blocking, T_z has to be $2 * B_k + 2$, where B_k is the iteration blocking factor. For 4-way blocking, an additional $B_z - 1$ planes have to be held in cache, B_z being the factor by which the Z-loop is blocked.

$$
\begin{aligned}
T_z &= 2^{\left\lceil \log_2 \sqrt{C/T_{minZ}} \right\rceil} \\
T_x &= 2^{\left\lceil \log_2 \sqrt{C/T_z} \right\rceil} \\
T_y &= C/(T_z T_x) \\
D_{Px} &= 2T_x \left\lfloor (D_x + 3T_x - 1)/(2T_x) \right\rfloor - T_x \\
D_{Py} &= 2T_y \left\lfloor (D_y + 3T_y - 1)/(2T_y) \right\rfloor - T_y
\end{aligned} \tag{6.1}
$$

With D_{Px} and D_{Py} the padded array dimensions have been calculated. To implement a non-standard padding as described in Section 1.3 the array can be declared with dimension D_y and padded by $(D_{Px} * (D_{Py} - D_y) \bmod C)$. For the resulting tile dimensions T_x,T_y and T_z, the size of the relaxation stencil has to be taken into account. The 7–point stencil of the reference implementation includes elements with distance 1 into the calculation. Thus 2 has to be subtracted from each tile dimension, which gives the blocking factors for all three loops as $(T_x - 2)$,$(T_y - 2)$, and $(T_z - 2)$.

6.2 Proof

The following proof has been proposed by G. Rivera[1]. It assumes a direct mapped cache, and thus also works for set-associative caches.

Given two references to different array elements in the same tile (T_x, T_y, T_z), let $D = d_x + d_y * S_x + d_z * S_x * S_y$, where d_x, d_y, d_z are the differences of the two points in X,Y and Z dimension, and S_x, S_y, S_z are the dimension of the array which is accessed. It has to be shown that $(D \bmod C) \neq 0, \forall d_i, d_i < T_i, i \in \{x, y, z\}$, simply meaning that no two different data items are allowed to be mapped to the same cache location.

Assumptions:

$$\begin{array}{rcl} A_1 & : & d_i < T_i \\ A_2 & : & not(d_x = d_y = d_z = 0) \\ A_3 & : & C = 2^k \\ A_4 & : & C = T_x * T_y * T_z \\ A_5 & : & S_x/T_x, S_y/T_y \text{ odd integers} \\ & & \text{which is equivalent to } GCD(C, S_i) = T_i \end{array} \tag{6.2}$$

For each line the used assumptions will be written in parentheses at the end of the line. In the following $\mathrm{RP}(x, y)$ will denote that x is relatively prime to y.

i) $d_x \neq 0$:

$$\begin{array}{rcl} d_x \bmod T_x & \neq & 0 \;\; (A_1, d_x \neq 0) \\ d_x + T_x * (d_y * (S_x/T_x) + d_z * (S_x/T_x) * S_y) \bmod T_x & \neq & 0 \\ D \bmod T_x & \neq & 0 \\ D \bmod C & \neq & 0 \;\; (A_4) \end{array}$$

ii) $d_x = 0, d_y \neq 0$:

$$\begin{array}{rcl} d_y \bmod T_y & \neq & 0 \;\; (A_1, d_y \neq 0) \\ d_y * T_x \bmod T_x * T_y & \neq & 0 \\ d_y * T_x * (S_x/T_x) \bmod T_x * T_y & \neq & 0 \;\; (A_3, A_4, \mathrm{RP}(S_x/T_x, T_x * T_y)) \\ d_y * S_y \bmod T_x * T_y & \neq & 0 \\ d_x + d_y * S_y \bmod T_x * T_y & \neq & 0 \;\; (d_x = 0) \\ d_x + d_y * S_x + T_x * T_y * (d_z * (S_x/T_x) * (S_y/T_y)) \bmod T_x * T_y & \neq & 0 \\ D \bmod T_x * T_y & \neq & 0 \\ D \bmod C & \neq & 0 \;\; (A_4) \end{array}$$

iii) $d_x = d_y = 0, d_y \neq 0$:

$$\begin{array}{rcl} d_z \bmod T_z & \neq & 0 \;\; (A_1, d_z \neq 0) \\ d_z * T_x * T_y \bmod T_x * T_y * T_z & \neq & 0 \\ d_z * T_x * T_y * (S_x/T_x) * (S_y/T_y) \bmod T_x * T_y * T_z & \neq & 0 \;\; (A_3, A_4, \mathrm{RP}((S_x/T_x) * (S_y/T_y), T_x * T_y * T_z)) \\ d_z * S_x * S_y \bmod T_x * T_y * T_z & \neq & 0 \end{array}$$

[1] This proof is previously unpublished, and thus shown here completely.

$$
\begin{array}{rcl}
d_z * S_x * S_y \bmod C & \neq & 0 \;\; (A_4) \\
d_x + d_y * S_x + d_z * S_x * S_y \bmod T_z & \neq & 0 \;\; (d_x = d_y = 0) \\
D \bmod C & \neq & 0 \;\; (A_4)
\end{array}
\tag{6.3}
$$

Note that Assumptions 1 to 5 are fulfilled, if Equation (6.1) is used to determine tile sizes and padding.

6.3 Results

6.3.1 For the reference multigrid program

Within the scope of this thesis, the padding approach was tested for the reference multigrid program with its two arrays, on Platform A optimizing for the level 2 cache, which is 96K on this machine. For different problem sizes, no performance gain could be measured in this case. This might not be unexpected, as Equation (6.1) does not include the number of involved arrays, which is a significant difference.

Furthermore, the method using Equation (6.1) is not transferable to this case, as the access patterns of the array of unknowns, and the array containing the right hand side and coefficients are different. The first one is accessed in spatial groups, while the second is accessed strictly serially. The right hand side and coefficients are never reused, except across different iterations, while each unknown is needed for the calculation of its six neighbors. So it is not efficient to apply Equation (6.1) to both arrays, as this does not avoid conflicting accesses to the different arrays. Still the unknowns require only a small part of the total memory (1/9 in this case) and thus the correct storage of the other arrays, inflicting as few conflicts as possible, is important.

Furthermore it is not possible to include any kind of cache memory distribution, to limit the space each array should require in cache. The cache organization algorithms can currently not be changed by an application, making it impossible to reserve a larger part of the cache for the unknowns, and a smaller one for the other arrays, for example.

It is not efficient either to provoke conflicts of the coefficients array to achieve this kind of memory distribution. This could be imagined to have a positive effect, as there is almost no reuse of these values, and provoking conflicts in this array would avoid any conflicts with the array of unknowns, resulting in a limited cache space where all of these values are stored. The poor performance in this case results from the conflict misses in a lower cache level (Level 3 in this case), as reuse there is still important, but the size of a lower cache level is usually a multiple of the lower one. This padding could be calculated as $P_X = k - D_X$, where P_X is the resulting padding in X-direction, D_X is the dimension of the array in and the cache size is a multiple of k: $C_2 = k * s_2$ (C_2 and C_3 are the sizes of the Level 2 and 3 Caches, s_2 and s_3 arbitrary integer values). So, if the coefficients array is padded to have conflict misses in this cache level, the lower level usually has the capacity $C_3 = k * s_2 * s_3$, and thus the accesses also lead to conflicts there.

6.3.2 For a homogenous problem

To evaluate the effects for a single array, the multigrid program without extra arrays for coefficients was used. The right hand side arrays are needed only for the coarser grids. In this case, the analytical padding with Equation (6.1) yields better results than no padding, but is not optimal.

	Running time in seconds	MFLOPS
Multigrid program without padding	9.2	63.95
Padding and tiling using Equation (6.1)	7.7	76.37
Padding and tiling from exhaustive search	6.8	86.48

It is certainly useful to avoid pathological cases, but the alignment of the data for a multigrid program is not that bad (the grids normally have a size of $2^k + 1$), and the non-conflicting arrangement for one cache level is not the only effect that determines the performance. The effects resulting from multiple cache levels also need to be taken into account, and the TLB cache can have a significant impact on the performance for larger tile sizes, and higher numbers of blocked iterations.

6.3.3 In general

Due to the large parameter space that needs to be searched for a blocked 3D Gauss-Seidel implementation, an analytical approach would be very useful. Still, all arrays needed for the computation need to be taken into account, as well as all levels of cache and the TLB, which makes it hard to find non-conflicting paddings. The TLB could also be included, by taking into account the page size and capacity, and then checking for a given tile size and padding, if the accesses lead to an overflow. This would also work easily for multiple arrays, while the prevention of conflicts for multiple arrays, that might have different access patterns, seems to require heuristical or empirical approaches. Still, for calculations on only a single array it is a very efficient way to rule out pathological cases.

Chapter 7

Conclusions

Several cache optimization techniques for a 3D multigrid application were evaluated in this thesis. It could be shown, that for an efficient program a cache-aware implementation and ordering of loops is definitely important. This includes a good data layout – for 3D, similar to 2D, an access-oriented data layout has shown to allow the best exploitation of spatial locality. The effect of padding varies on different platforms, but can increase the performance up to 60%, while architectures like the Pentium 4 used in Platform F show a smaller increase. The most efficient blocking techniques also vary from platform to platform, but, up to the 1-way blocking, achieve good results. The more complex blocking variants are more efficient for bigger grids, and other effects like TLB cache misses start dominating the performance. To avoid these, data copying techniques, or the calculation of maximum tile sizes, still have to be evaluated.

The analytical padding approach has shown to be inapplicable to the 3D multigrid program, other ways of integration should be investigated, as exhaustive searches for 3D can take huge amounts of time due to the large parameter space.

While blocking and padding variants need to be tailored to compiler and architecture, be it automatically or by hand, the multigrid algorithm itself can also be adapted more to the cache architectures of modern computers. The use of patch-adaptive multigrid algorithms [LR97] or on the fly calculation of matrix entries, for example, are current research projects.

Appendix A

Machine Specifications

A.1 Platform A

- CPU: Alpha 21164
- Workstation: DEC PWS 500au
- Name: fauia31.informatik.uni-erlangen.de
- L1 Cache: 8KB Data, 8KB Instructions, direct mapped, line size 32 bytes
- L2 Cache: 96KB, 3-way set-associative, line size 32/64 bytes
- L3 Cache: 4MB, direct-mapped, off chip module
- TLB Size: 128 entries
- Operating system: DIGITAL UNIX 4.0D
- Compiler: DEC Fortran77 5.1
- Compiler flags: `-extend_source -fast -O4 -pipeline -tune host -arch host -g3`

A.2 Platform B

- CPU: Alpha 21164
- Workstation: DEC PWS 500au
- Name: fauia30.informatik.uni-erlangen.de
- L1 Cache: 8KB Data, 8KB Instructions, direct mapped, line size 32 bytes
- L2 Cache: 96KB, 3-way set-associative, line size 32/64 bytes
- L3 Cache: 4MB, direct-mapped, off chip module
- TLB Size: 128 entries
- Operating system: Compaq Tru64 UNIX V5.0A
- Compiler: DEC Fortran90 5.3
- Compiler flags: `-extend_source -fast -O4 -pipeline -tune host -arch host -g3`

A.3 Platform C

- CPU: Alpha 21264
- Workstation: Compaq XP 1000 professional
- Name: fauia35.informatik.uni-erlangen.de
- L1 Cache: 64KB Data, 64KB Instructions, 2-way set-associative, line size 32 bytes
- L2 Cache: 4MB, direct-mapped, off chip
- TLB Size: 128 entries
- Operating System: Digital UNIX V4.0F
- Compiler: DEC Fortran77 5.2
- Compiler flags: `-extend_source -fast -O4 -pipeline -tune host -arch host -g3`

A.4 Platform D

- CPU: AMD Athlon 700MHz
- Name: fauia28.informatik.uni-erlangen.de
- L1 Cache: 64KB Data, 64KB Instructions, 2-way set-associative
- L2 Cache: 256KB, direct-mapped, off chip
- TLB Size: 32 entries L1-TLB, 256 entries L2-TLB
- Operating System: Linux with Kernel 2.4.10
- Compiler: GNU Fortran77 2.95.3
- Compiler flags: `-O3 -fforce-addr -ffixed-line-length-none`

A.5 Platform E

- CPU: AMD Athlon 700MHz
- Name: fauia25.informatik.uni-erlangen.de
- L1 Cache: 64KB Data, 64KB Instructions, 2-way set-associative
- L2 Cache: 256KB, direct-mapped, off chip
- TLB Size: 32 entries L1-TLB, 256 entries L2-TLB
- Operating System: Linux with Kernel 2.4.10
- Compiler: GNU Fortran77 2.91.66
- Compiler flags: `-O3 -fforce-addr -ffixed-line-length-none`

A.6 Platform F

- CPU: Intel Pentium 4 1.5GHz
- Name: fauia43.informatik.uni-erlangen.de
- L1 Cache: 8KB Data
- L2 Cache: 256KB
- TLB Size: 64 entries
- Operating System: Linux with Kernel 2.4.10
- Compiler: GNU Fortran77 2.95.2
- Compiler flags: `-O3 -fforce-addr -ffixed-line-length-none`

A.7 Platform G

- CPU: Intel Celeron 366MHz
- L1 Cache: 16KB
- L2 Cache: 128KB
- TLB Size: 32 entries
- Operating System: Linux with Kernel 2.4.10
- Compiler: GNU Fortran77 2.95.3
- Compiler flags: `-O3 -fforce-addr -ffixed-line-length-none`

Appendix B

3D Multigrid Implementation

Description of red/black Gauss-Seidel subroutines for the Fortran77 3D multigrid implementation:

- `rbszyx`: Standard red/black Gauss-Seidel implementation with two loop nests and loop order K-Z-Y-X.
- `rbszxy`: The same as rbszyx but with loop order K-Z-X-Y.
- `rbsyzx`: The same as rbszyx but with loop order K-Y-Z-X.
- `rbsyxz`: The same as rbszyx but with loop order K-Y-X-Z.
- `rbsxzy`: The same as rbszyx but with loop order K-X-Z-Y.
- `rbsxyz`: The same as rbszyx but with loop order K-X-Y-Z.
- `rbfuse`: Fused implementation.
- `rbtile`: 2-way blocked X- and Y- loops, no blocking of iterations.
- `rb1w`: 1-way blocked red/black Gauss-Seidel with loop order K-Z-Y-X-Bk
- `rb1wli`: Further loop interchange resulting in loop order K-Z-Y-Bk-X.
- `rb1wo2`: The same as rb1w but hand unrolled for two blocked iterations.
- `rb2w`: 2-way blocked, loop order K-Z-Y-X-Bk-Bx.
- `rb3w`: Blocked iteration,X- and Y- loops.
- `rb4w`: All four loops blocked.

List of Figures

Bibliography

[ABD+97] 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.

[BDQ98] F. Bassetti, K. Davis, and D. Quinlan. Temporal Locality Optimizations for Stencil Operations within Parallel Object–Oriented Scientific Frameworks on Cache–Based Architectures. In *Proc. of the International Conf. on Parallel and Distributed Computing and Systems*, pages 145–153, Las Vegas, Nevada, USA, October 1998.

[BHM00] W.L. Briggs, V.E. Henson, and S.F. McCormick. *A Multigrid Tutorial*. SIAM, 2000.

[DHK+00] C.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, February 2000.

[Dou96] C.C. Douglas. Caching in With Multigrid Algorithms: Problems in Two Dimensions. *Parallel Algorithms and Applications*, 9:195–204, 1996.

[DS98] K. Dowd and C. Severance. *High Performance Computing*. O'Reilly, 1998.

[FJ98] M. Frigo and S.G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP'98)*, volume 3, pages 1381–1384, May 1998.

[GJF01] J. Mellor-Crummey G. Jin and R. Fowler. Increasing Temproal Locality with Skewed and Recursive Blocking. In *Proceedings of the Supercomputing Conference 2001*, Denver, November 2001.

[GKKS01] W.D. Gropp, D.K. Kaushik, D.E. Keyes, and B.F. Smith. High Performance Parallel Implicit CFD. *Parallel Computing*, 27(4):337–362, March 2001.

[Han98] J. Handy. *The Cache Memory Book*. Academic Press, second edition, 1998.

[HP96] J.L. Hennessy and D.A. Patterson. *Computer Architecture — A Quantitative Approach*. Morgan Kaufmann Publishers, second edition, 1996.

[KWR02] M. Kowarschik, C. Weiß, and U. Rüde. Data Layout Optimizations for Variable Coefficient Multigrid. In *Proceedings of the 2002 International Conference on Computational Science*, Lecture Notes in Computer Science, Amsterdam, The Netherlands, April 2002. Springer. to appear.

[Los98] D. Loshin. *Efficient Memory Programming*. McGraw–Hill, 1998.

[LR97] H. Lötzbeyer and U. Rüde. Patch–Adaptive Multilevel Iteration. *BIT*, 37(3):739–758, 1997.

[MLW91] E. Rothberg M.S. Lam and M.E. Wolf. The Cache Performance and Optimization of Blocked Algorithms. In *Fourth Intern. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, Palo Alto, California, April 1991.

[Pfä00] H. Pfänder. Cache–optimierte Mehrgitterverfahren mit variablen Koeffizienten auf strukturierten Gittern. Master's thesis, Department of Computer Science, University of Erlangen–Nuremberg Germany, 2000.

[Riv01] G. Rivera. *Compiler Optimizations for Avoiding Cache Conflict Misses*. PhD thesis, Dept. of Computer Science, University of Maryland, College Park, MD, USA, 2001.

[RT00] G. Rivera and C.-W. Tseng. Tiling Optimizations for 3D Scientific Computations. In *Proceedings of the ACM/IEEE Supercomputing 2000 Conference (SC2000)*, Dallas, TX, USA, November 2000.

[SL99] Y. Song and Z. Li. New Tiling techniques to improve Cache Temporal Locality. In *ACM SIGPLAN PLDI99*, 1999.

[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 the ACM/IEEE SC93 Conference*, Portland, OR, USA, November 1993.

[TOS01] U. Trottenberg, C. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, 2001.

[WD98] R.C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proceedings of the International Conference on Supercomputing*, Orlando, Florida, USA, November 1998.

[Wei01] C. Weiß. *Data Locality Optimizations for Multigrid Methods on Structured Grids*. PhD thesis, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik, Technische Universität München, Munich, Germany, December 2001.

[WKKR99] C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory Characteristics of Iterative Methods. In *Proceedings of the ACM/IEEE SC99 Conference*, Portland, Oregon, November 1999.

[Yav96] Irad Yavneh. On Red-Black Sor smoothing in Multigrid. In *SIAM J. SCI. COMPUT.*, volume 17, pages 180–192, January 1996.

[ZZ99] Z. Zhang and X. Zhang. Cache-Optimal Methods for Bit-Reversals. In *Proceedings of the Supercomputing Conference 1999*, Portland, Oregon, November 1999.