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

Lehrstuhl für Informatik 10 (Systemsimulation)

Cache–optimierte Mehrgitterverfahren mit variablen Koeffizienten auf strukturierten Gittern

Harald Pfänder

Diplomarbeit

Cache–optimierte Mehrgitterverfahren mit variablen Koeffizienten auf strukturierten Gittern

Harald Pfänder

Diplomarbeit

Aufgabensteller:	Prof. Dr. Ulrich Rüde
Betreuer:	Dipl.–Inf. Markus Kowarschik
Bearbeitungszeitraum:	2. August – 2. Januar

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 28. Oktober 2003 ....

Contents

List of Figures

Zusammenfassung

Bei der numerischen Behandlung technisch–wissenschaftlicher Problemstellungen mittels iterativer Algorithmen kann oft die maximale Rechenleistung heutiger Prozessoren nur zu einem Teil genutzt werden. Eine Hauptursache dafür ist in der im Vergleich zur CPU–Leistung langsamen Speicheranbindung der meisten Prozessoren zu finden. Verursacht die CPU einen Speicherzugriff, müssen die Daten oft erst aus dem Hauptspeicher geladen werden. Bei der Lösung von Problemen mit großen Datenaufkommen wird eine beträchtliche Zeit nur mit dem Warten auf diese Daten verbracht. Aus diesem Grund sind die Speichersysteme heutiger CPUs hierarchisch aufgebaut, um diese Zugriffszeiten zu verbergen. Das Ziel dieser Arbeit ist die Implementation und Untersuchung von Cache–optimierten Mehrgitterverfahren mit variablen Koeffizienten. Um die Programmlaufzeiten für solche Problemgrößen niedrig zu halten, ist es deshalb wichtig, möglichst wenige Hauptspeicherzugriffe und viele Zugriffe auf Cache–Inhalte zu erzeugen. Dies kann durch eine effiziente Datenhaltung erreicht werden, was Ziel dieser Arbeit ist.

Abstract

The solution of technical and scientific problems using iterative methods often can only reach a small percentage of the peak performance of today's processors. One main cause of this can be found with the low memory performance of the most processors in comparison to the CPU performance itself. If the CPU causes a memory access, all data have to be fetched from main memory. With the solution of problems with large amounts of data a long period of time has to be spent waiting for these data. For this reason the memories of today's CPUs are structured hierarchically in order to hide these access times. The goal of this work is the implementation and investigation of cache–optimized multigrid methods with variable coefficients. In order to shorten the execution times for these problem sizes it is important to produce preferably fewer accesses to main memory and a lot of accesses to the cache contents. This can be achieved using an efficient memory storage, which is the goal of this work.

Chapter 1

Introduction

In this work we investigate the cache behavior of multigrid methods with variable coefficients. The research is motivated by the fact that on the one hand, the numerical solution of *partial differential equations* (PDEs) is required in many technical and scientific simulations. This often leads to large systems of linear equations. Multigrid algorithms [Bri00] have been shown to be among the most efficient methods for solving such large linear systems.

On the other hand in today's CPU architectures one of the the most speed limiting factors is the cost of accesses to memory. Every reference to main memory can be seen — in contrast to the execution times of arithmetic operations as very time–consuming. In the last years usually the performance of the processors has doubled every year. In comparison, the memory performance has doubled every seven years.

From this the performance suffers if very large amounts of data must be fetched from memory. This represents a serious bottleneck for high performance computing. For this reason most workstation architectures provide a hierarchal memory organization. For caches that are located closer to the CPU, the references are much faster than the main memory accesses. If data accesses can be made more local these cache levels can be used more efficiently.

With the growing performance of most CPUs also the opportunity to solve more complex problems with larger data sizes increases. For most of today's problems the amount of data is usually too large to fit completely into the cache levels of the CPU. Thus, if significant parts of the cache contents can be reused, this would lead to a performance gain.

The iterative methods that generally are used within a multigrid algorithm also provide some kind of data locality, e.g. if all unknowns are referenced in sequential order. Another approach to improve the cache efficiency is the use of *working sets*. In fact, the cost for the collection of data from the memory to the cache levels is high, but if these cache contents can be reused multiple times the references to these locations can be done much faster. In [HP96] the author specifies the cache access time about 3–10 nano seconds whereas the main memory access time is specified about 80–400 nano seconds. For this also the size of the working set must be chosen in a way that it fits into the cache to benefit from this effect.

In this thesis we investigate the impact of cache optimizations for multigrid algorithms with variable coefficients on structured grids. In comparison to constant coefficient problems this provides more flexibility and generality concerning the problems that are modeled. One of our model problems including strong singularities can be found at [Rüd81].

However, the use of variable coefficients also increases the amount of data that has to be stored for each unknown, in comparison to algorithms with constant coefficients. Therefore data layout methods must be considered in order to achieve high data locality. Also various optimization methods must be applied in order to investigate their impact on the cache performance.

The main focus of this work is on the implementation and research of optimization methods for 2–D multigrid methods. Therefore we developed a 2–D multigrid code on structured rectangular grids involving Dirichlet boundary conditions.

The program has been implemented in the programming language C. The solution on the coarsest grid can be obtained with the help of a direct solver. For that the *LAPACK* library has been integrated. For the results all optimizations are applied to the our code. With the performance test an evaluation of the results is done with a special focus on cache efficiency enhancements.

In Chapter 2 we give a brief overview of the mathematical backgrounds including PDEs and multigrid methods. In Chapter 3 we explain the memory hierarchy and the architectures with which our performance measurements were done. In Chapter 4 the program implementation is explained including a short overview of the *LAPACK* library and the profiling tools *PCL* and *DCPI* which are mainly used. In Chapter 5 the optimization methods that are applied are explained in detail. For each optimization variant we show how the performance could be improved. Therefore we have done performance measurements on several architectures to verify these results and to show the impact of these optimizations on different architectures. The chapter ends with a case study on the Alpha 21164 where a more detailed profiling example is shown for our multigrid code. Chapter 6 draws our final conclusions.

Chapter 2

Mathematical Basics

In this chapter the mathematical aspects of multigrid algorithms are described briefly. We therefore outline some basic facts about the numerical solution of *partial differential equations* (PDEs) in the first section. In the next section we give an example of a multigrid algorithm and show its functionality. For a more detailed overview consult e.g. [Bri00, Sch97, ST82].

2.1 Partial Differential Equations

Scientific computing often requires the numerical solution of PDEs. These equations often occur e.g. in chemistry, physics, computational fluid dynamics, electrical engineering, etc.

In this work we focus on elliptic second–order PDEs of the general form:

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Du_x + Eu_y + Fu = H \tag{2.1}$$

where

$$AC - B^2 > 0,\ \forall (x, y) \in G \tag{2.2}$$

holds.

The variables A, B, C, D, E, F, H in equation (2.1) may be continuous functions, G specifies the domain in which the PDE is defined.

One example of an elliptic second–order PDE is *Poisson*'s equation:

$$-\triangle u = f \tag{2.3}$$

This important PDE e.g. describes the steady–state temperature distribution in a homogeneous medium.

In addition to the PDE itself, boundary information is needed. As usual, we use the symbol ∂G to denote the boundary of G. Examples for boundary conditions are:

$$\begin{aligned} u &= \phi \text{ on } \partial G && \text{(Dirichlet-Condition)} \\ \frac{\partial u}{\partial n} &= \varphi \text{ on } \partial G && \text{(Neumann–Condition)} \\ \frac{\partial u}{\partial n} + \alpha u &= \beta \text{ on } \partial G && \text{(Cauchy–Condition)} \end{aligned} \tag{2.4}$$

Mixed boundary conditions are also possible. The variables ϕ, φ, α in equation (2.4) mark known functions. In our implementation we only concentrate on Dirichlet boundary conditions:

$$u = \phi \text{ on } G$$

The numerical solution of PDEs is usually based on discretization techniques. There are different possibilities for the discretization method, e.g. finite elements, finite volumes and finite differences. In our work we focus on finite differences which can e.g. be found in [MG80].

For the solution with finite differences the domain G, in which the PDE is defined, is discretized using a regular grid. This method is shown in Figure 2.1.

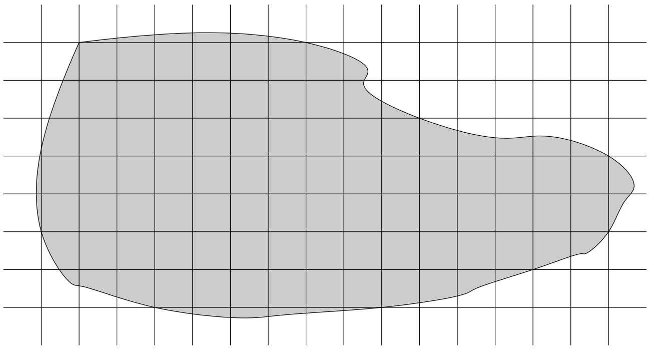


Figure 2.1: The domain G; it is discretized using a structured grid

For the example in this figure we used a fixed mesh size in both directions. The resulting grid can be labeled ***structured***. In the course of the following numerical solution process, we only consider the continuous solution of the PDE at the discrete points of the grid, respecting the given boundary conditions.

At each inner grid point, the partial derivatives occurring in the PDE are represented by finite difference approximations, e.g.

$$u_{xx}(x_i, y_i) \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} \tag{2.5}$$

It is often useful to introduce the following notation:

$$u_{i+1,j} = u_E, u_{i-1,j} = u_W, u_{i,j+1} = u_N, u_{i,j-1} = u_S, u_{i,j} = u_{CE} \tag{2.6}$$

If the technique is applied to Poisson's equation we receive the following approximation for each inner grid point.

$$4u_{CE} - u_E - u_W - u_N - u_S - h^2 f = 0 \tag{2.7}$$

In this example each point is calculated using its own value and the values of the four neighbors in the north, south, east and west direction. The approximate solution for each point thus is represented using a *5–point–stencil*. Alternatively, discretizations using compact 9–point–stencils could also be applied [Sch97].

When the approximation for each unknown is done the following discretized equation occurs for the computation of the solution vector $\vec{u}_h$:

$$A_h \vec{u_h} = \vec{f_h} \tag{2.8}$$

The matrix A_h consists of all coefficients, $\vec{u}_h$ contains the unknown values and $\vec{f_h}$ is the right–hand–side (rhs) also including the boundary values of $\vec{u}_h$. Thus we receive one equation for each unknown. The coefficient matrix that is used inside this thesis uses a 5–point–stencil with variable coefficients. This means that the matrix entries may differ from unknown to unknown. This provides more flexibility and generality than an implementation based on constant coefficients, for example.

2.2 Multigrid Approach

For the numerical solution of linear systems of equations — as shown in the previous section — multigrid methods can be used. These methods usually have the advantage of better and faster convergence rates in comparison to the most direct solvers or basic iterative methods. For a more detailed discussion we refer to [Bra77, Bra84, Hac85].

A multigrid algorithm consists of more components that work together. On the different levels an iterative solver is used to smooth the current approximate solution. An error correction is recursively computed on the next coarser level with which the approximated solution is then corrected.

In our multigrid implementation we use the method of Gauss–Seidel to smooth the approximations on each grid level.

The Gauss–Seidel method is defined as follows:

$$u_i^{k+1} = -\frac{1}{a_{i,i}}\left(\sum_{j=1}^{i-1} a_{i,j}u_j^{k+1} + \sum_{i=1+1}^{n} a_{i,j}u_j^{k} - f\right), \tag{2.9}$$
$$A = a(i,j)_{1\leq i,j\leq n}, \vec{u} = (u_1, u_2, \ldots, u_n)^\top, k \geq 0$$

In our multigrid method we use a *red–black* ordering of the unknowns. In a red–black smoother all elements inside the 2–D field are colored red and black. Figure 2.2 shows a 2–D grid with five intervals in each direction. Note that we assume Dirichlet boundary conditions where the boundary values are fixed.

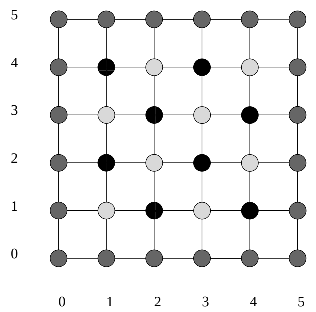


Figure 2.2: A 2–D grid with 5x5 grid nodes colored red–black.

One advantage of this red–black smoother is that there are only data–dependencies between the red and black points. In one iteration sweep through the grid all red points must be relaxed first. Each black point which is to be relaxed needs the newly computed solution values at its neighboring red points. So with this method we must care about the data–dependencies between red and black points.

The multigrid method additionally transports information between the grids. This can be done in different ways. Usually one uses a cycle scheme to switch between the different grids. The information can be restricted to coarser grids with an *restriction operator* and prolongated to finer grids with an *interpolation* or *prolongation operator*. In this thesis we implemented a standard *V–cycle* scheme with is explained in detail in [Bri00].

In our V–cycle scheme, first the approximation is smoothed with several presmoothing steps — given with μ_1 — using the red–black Gauss–Seidel algorithm. After that a correction is computed on the coarser grids. Therefore the residual equation is approximately solved on the next coarser level.

The residual again can be transfered with different methods. In our thesis we use *full–weighting* as restriction operator. In stencil notation, the restriction operator can be defined as follows:

$$I_h^{2h} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{2.10}$$

After the residual has been restricted to the coarser grid, the error correction can be computed:

$$A_{2h}\vec{e}_{2h} = \vec{r}_{2h} \tag{2.11}$$

This problem is a coarse grid representation of the fine grid residual equation. This equation can be solved recursively until the coarsest grid is reached. In our multigrid implementation we restrict until the coarsest grid consists of only one unknown. Then the equation can be solved directly with one Gauss–Seidel step. Additionally we also implemented a direct solver using the *LAPACK* library — which is described in Section 4.3 . For this enhancement the coarsest grid can also consist of more unknowns.

The choice of the coarse grid coefficients can also be done in different ways. One way is to discretize the PDE on each grid level. This is what we do.

After the error vector has been computed on the coarser level it is prolongated to the finer level. For the prolongation method also many methods could be used. One method for prolongation is the linear interpolation. The linear interpolation operator is defined as follows:

$$I_{2h}^h = \frac{1}{4}\left]\begin{matrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{matrix}\right[\tag{2.12}$$

With this method the approximate solution $\vec{v}_h$ on the finer level is corrected with the error vector $\vec{e}_{2h}$ on the coarser level. At the end of this V–cycle scheme additional postsmoothing can be done. The number of postsmoothing steps is given as μ_2.

In order to show the different convergence rates of a multigrid algorithm and a basic iterative solver we illustrate one simple example for the solution of a PDE with our implemented multigrid algorithm. For this Poisson model problem we use the following initialization:

$$\begin{aligned} u_h &= 5\sin(\pi x)\sin(\pi y) + \sin(15\pi x)\sin(15\pi y) \\ f_h &= 0 \\ u &= 0 \text{ on } \partial G \text{ (Dirichlet boundary condition)} \\ n &= 32 \text{ (number of grid intervals)} \end{aligned}$$

For this model example the trivial solution $\vec{u} = 0$ is known. In Figure 2.4 the output of one V–cycle is plotted. In the upper left figure one can see the initial solution $\vec{u}_h$. In the upper right figure two red–black Gauss–Seidel steps were computed as presmoothing steps. The lower right figure shows the approximate solution after the error correction from the coarser grid is done. The lower right figure shows the approximate solution after the two postsmoothing steps.

In Figure 2.5 we have illustrated the approximate solution vector after 100 red–black Gauss–Seidel iterations. There we can see in contrast to the multigrid V–cycle that the convergence rate is lower

Number of floating points	
Gauss–Seidel algorithm (100 iterations)	961.000
Multigrid method (1 V–cycle, $\mu_1 = 2, \mu_2 = 2$)	70.502

Figure 2.3: The number of floating–point operations that are needed for 100 Gauss–Seidel iterations in contrast to one complete V–cycle.

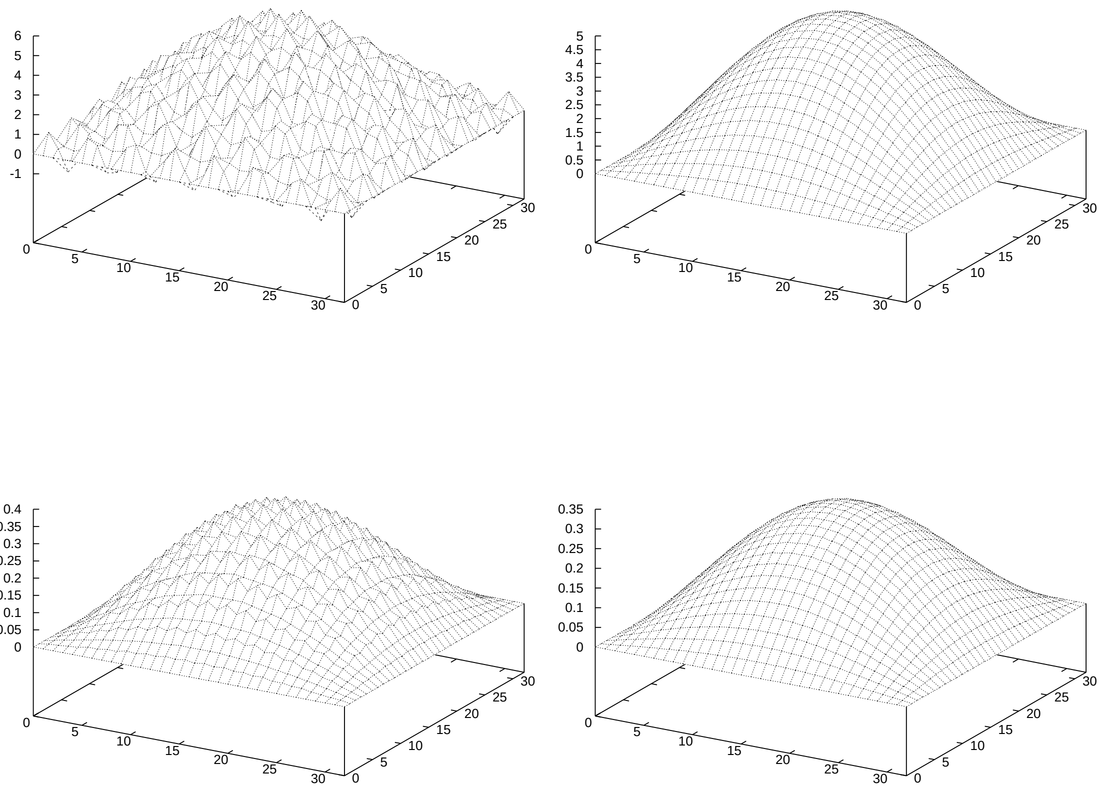


Figure 2.4: The approximate solution for one V–cycle of the multigrid algorithm. The initial approximation (upper left), the solution vector after two red–black Gauss–Seidel presmoothing steps (upper right), the solution vector after the coarse–grid correction (lower left) and the solution vector after two postsmoothing steps.

for this model problem. This is due to the fact that also the low–frequency error parts are reduced fast with the multigrid algorithm. The Gauss–Seidel method is much slower for smoothing the low–frequency error components. The high–frequency components in contrast can be eliminated faster. In Table 2.3 we have compared the number of floating–point operations that were needed to compute this model problem with the Gauss–Seidel smoother and with one multigrid V–cycle. There we can see that for the multigrid algorithm about 10 times fewer floating–point operations are needed than for the Gauss–Seidel smoother with even a much better convergence rate. For this the multigrid method usually is more attractive for the solution of large linear systems of equations.

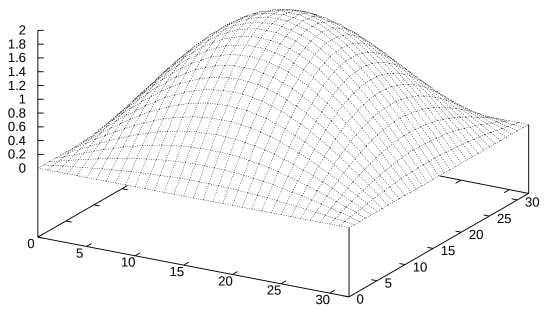


Figure 2.5: The approximate solution for $\vec{u}_h$ after 100 Gauss–Seidel iterations.

Chapter 3

Architecture Overview

In this chapter, we briefly describe the different architectures which were used in this work. To underline the impact of cache–efficient programming we explain the general structure of memory hierarchies, which can be found in most of today's architectures.

A more detailed overview of basic architecture aspects can be found in [DS99, HP96].

3.1 Memory Hierarchy

The memory requirements for today's problems are very high and also the main memory provides the bottleneck for most programs that can be found in high performance computing. The goal therefore is to implement fast and big memory. Because the cost for this are very high, computer designers decide to implement the memory in a hierarchical order — as shown in the Figure 3.1 — in order to find a compromise between cost and efficiency. In the figure the memories on the top level are the fastest. With descending level the size of memory grows but with it also the bandwidth and the speed decrease.

The fast and expensive CPU registers can usually be found in the top of this memory pyramid. The registers can be accessed very fast but they are also very expensive. This is due to the fact that they are implemented directly into the processor so that the bandwidth also is extremely high. From this reason the number of registers usually is very low in comparison to the amount of main memory. Also the space on the processor die is rare and with the amount of registers also the size of the processor grows.

In the levels below the caches are usually found. In the figure we have illustrated two cache levels.

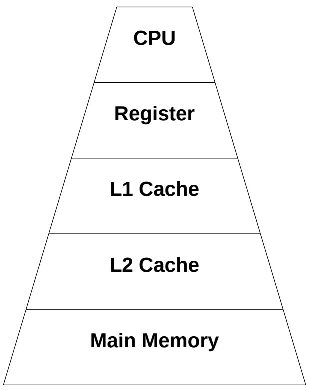


Figure 3.1: The memory structure which usually can be found in today's architectures.

The caches are slower than the registers but faster than main memory. The capacity of the caches generally is much bigger than the capacity of the registers. The size, the number of levels and the place depends on various design studies of the different manufactures and also on the requirement fields that the processors have. Mostly the L1 cache is implemented on chip. The L2 cache can be implemented as off chip cache. The Intel Pentium II and the AMD Athlon have implemented the L2 cache off chip. In contrast, the Intel Pentium III and the AMD Athon/Thunderbird have implemented the L2 cache on chip. The HP PA85000 has only one cache level and the Alpha 21164 in contrast has three cache levels. On chip caches are generally faster than off chip caches and provide a higher bandwidth because they are physical closer to the CPU and therefore can be accessed in a more faster way.

In the last section of the memory structure the main memory can be found. Apart from that also other memory storage can be found e.g. the hard disk or magnetic tapes. But for this thesis only the correlation between main memory and the faster memory levels are investigated. The main memory in difference to the caches and the registers is slow but provides the largest amount of space.

3.2 Processor Types

This section describes the architectures on which we performed our experiments. Due to the fact that this thesis does not study the differences between several architectures, but rather the influence the cache sizes and memory behavior have on the execution time, only some special values of interest — such as cache size, CPU speed (MHz rate) and main memory size — are listed for the different architectures. In advance, Figure 3.2 provides a quick overview of the different cache levels and cache sizes the CPUs under consideration have.

For the measurements it is also important to state that, on all x86 architectures, Linux (with kernel version 2.2.16) was installed. The used compiler was gcc, version 2.35. The Alpha architectures are running under Digital Tru64 Unix, versions 4.0D and 4.0E, with the Digital cc compiler.

3.2.1 Intel Pentium II, Celeron, Xeon

The Pentium II provided for the measurements worked at a clock rate of 350 MHz, the Celeron had a speed of 400 MHz. Both systems had 128 MB of main memory. The specifications of both CPUs (Pentium II and Celeron) differ mainly in the size of the L2 cache. Both chips have a split 2–way L1 cache which is implemented as 4–way set–associative. The L1 cache is split into 16 KB data and into 16 KB instruction cache with a 32–byte cache line size.

The L2 cache is implemented as a unified cache for both CPUs and has a size of 512 KB for the Pentium II and 128 KB for the Celeron processor. Furthermore the L2 cache also works as 4–way associative with a 32–byte cache line size. The L2 cache is off chip and unified while the L1 cache is dual ported and integrated into the CPU. The Xeon processor in contrast provides the biggest L2 cache. The Xeon which was available for our measurements had a L2 cache size of 1MB, but the Xeon is also available with other L2 cache sizes. The specifications for the Intel CPUs were taken

	L1	L2	L3
Celeron (400)	32 KB	128 KB	
Pentium II (350)	32 KB	512 KB	
Xeon (700)	32 KB	1024 KB	
Athlon (700)	128 KB	512 KB	
Alpha 21164 (500au)	16 KB	96 KB	4MB
Alpha 21264 (XP1000)	128 KB	4 MB	

Figure 3.2: Cache sizes of the the CPUs on which the performance measurements were performed.

from the Intel website, see [Int99a, Int99b].

3.2.2 AMD Athlon

The machines with the AMD Athlon CPU worked at at clock rate of 700 MHz. We used a system with 128 MB of memory installed.

The Athlon has a dual ported 128 KB split L1 cache. It is separated into a 64 KB data and a 64 KB instruction cache. The L1 cache is 2–way associative both for data and instructions.

The L2 cache is located off chip, its size is 512 KB, and it is direct–mapped. The AMD Athlon processor has a super–scalar fully pipelined, out–of–order, three–way floating–point engine. This leads to about 1 GFLOPS of peek performance with double–precision instructions. The floating–point pipeline includes 15 stages.

The system bus works with 200 MHz and has about 1.6 GB per second of bandwidth. The branch prediction is implemented two–way with a 2048–entry branch prediction table. The Athlon also has multiple decoders that translate x86 instructions in *MacroOPs*. This means that all x86 instructions which can be of variable length (1 to 15 bytes) were translated in fixed length *MacroOPs*. The Athlon specification was taken form the AMD homepage, see [AMD99] for further details.

3.2.3 Alpha 21164, 21264

From the alpha architectures two different models were used for the experiments. We used a DEC PWS 500au and Compaq XP 1000 professional workstation for our measurements.

The DEC PWS 500au workstation contains the Alpha 21164 chip with 500 MHz. The CPU has three on chip caches: a primary L1 data cache, a primary L1 instruction cache and second level L2 cache. Both L1 caches have a size of 8 KB each, they are direct–mapped and their block line length is 32 bytes. The L1 data cache is dual–read–ported, single–write–ported and it is a write–through, read–allocate, direct–mapped, physical cache. In contrast to this, the L1 instruction cache is a virtual cache.

The L2 cache is a mixed data and instruction cache with a block line of 32 or 64 bytes. The L2 cache size is 96 KB and works as 3–way set–associative, physical, write back and write allocate. The L3 cache is off chip, direct–mapped, physical, write–back and write–allocate. The L3 cache size of our architecture was 4 MB. The cache read/write speed is 4 to 15 cycles, L1 cache hit is 2 cycles, L2 cache hit is ≥ 8 cycles.

The 21164 CPU has two integer pipelines, a floating–point multiply and a floating–point add pipeline. Therefore the 21164 CPU has a floating–point peek performance of 1 GFLOPS. The latency for a FDIV instruction is data dependent; 15 to 31 cycles for single precision and 22 to 60 cycles for double precision. The *latency* is defined as the number of cycles it takes between an instruction that produces an output and another instruction that can use the computed output. The Alpha 21164 also can have up to six outstanding loads. The main memory size was 640 MB. The resources for the DEC PWS 500au can be found at [Com96].

The Compaq XP 1000 includes the Alpha 21264 (EV6) processor running at 500 MHz. The CPU has two cache levels with a primary 64 KB L1 data cache and a 64 KB L1 instruction cache. The L2 cache is off chip with a capacity of 4 MB. The L1 instruction cache is virtually addressed and the line size is 32 bytes. The L1 data cache is virtually indexed, physically tagged and dual–read–ported with a 64 bytes line size. Both L1 caches are two–way set–associative. The L1 cache latency is 2–4 cycles. The L2 cache is direct–mapped, shared for both instruction and data and its latency is 12–14 cycles with a cache line size of 64 bytes. The L2 cache for our machine was 4 MB in size and the main memory size was 640 MB. The documentation for the XP 1000 can be found at [Com99].

3.2.4 Hitachi SR8000 (LRZ Munich)

The Hitachi SR8000–F1 currently consists of 112 *SMP* nodes. These symmetric massively parallel nodes can be accessed from a job queue. Every node consists of eight Power-3[1] RISC–processors with 8 GB main memory. The memory of the whole system is 928 GB.

The processors worked with a clock speed of 375 MHz. Each processor has a 64 KB two–way set–associative instruction cache. The data cache is 128 KB of size with a cache line size of 128 bytes. The data cache is implemented as a write–through cache and it is four–way set–associative. Both caches are implemented on chip. Every processor has 160 floating–point registers and 32 integer registers.

For our measurements we reserved one node exclusively and computed the results using one CPU, because our multigrid program is implemented as a sequential program.

To cover the main memory latency the Hitachi SR8000 is covered with unrolling, software pipelining and instruction prefetching.

Due to instabilities of the C–compiler not all of the measurements could be done, which prohibits a more detailed analysis of the cache behavior for this architecture. Nevertheless a few performance results about 2–D blocking can be found in Chapter 5.

A more detailed overview of the Hitachi SR8000 can be found at [LRM00].

3.2.5 HP PA8500

The PA–8500 processor from Hewlett–Packard differs from the other processors in that there exists only one cache level. The CPU works with a clock speed of 440 MHz. The PA–8500 CPU comprises a 1.5 MB cache which is on chip with a cache latency of about 2 cycles. The cache is divided in a 1 MB data and a 0.5 MB instruction cache.

The data cache is divided in 0.5 MB data cache banks, each of which is implemented as four one–eighth megabyte arrays. The data is organized within these arrays so that a full cache line can be addressed at one time or four ways of associativity can be addressed together.

The instruction cache is a four–way set–associative pipelined cache with a cache line size of 128 bytes.

The information was taken from [Pac98].

[1] The processor was additionally modified for the Hitachi SR8000.

Chapter 4

Program Organization

4.1 Overview

In this chapter we describe the implementation of our multigrid code. Since the major goal of this work is to develop and analyze cache optimization techniques for multigrid algorithms, we limited the complexity of the program. The implemented algorithm can handle 2–D problems with structured grids. Due to the fact that linear equations with variable coefficients are used the code in principle is able to handle problems with varying mesh widths.

In the following we first explain the structure of our 2–D multigrid code including all functions. The next section covers the enhancements and optimizations, that are applied to the program. The chapter ends with the profiling section. There, the main profiling tools that are used to analyze the structure of the program are explained. The chapter ends with a profiling experiment in order to show where the program has its hot spots and therefore where optimization should concentrate on.

4.2 Description of Functions

Our multigrid program is split into different functions. To compute a complete multigrid V–cycle different steps need to be done. At each level a smooth approximate solution is computed with an iterative solver. The residual of the smooth approximate solution is restricted to the coarser level and solved recursively in order to compute an error correction.

At the beginning of the multigrid program the different grids for each level have to be initialized. For our measurements we initialized the solution vector $\vec{u}_h^{(0)}$ with a constant value. Since we assume Dirichlet boundary conditions, we fix the values at the boundary grid points. At each unknown a 5–point–stencil with variable coefficients and a constant mesh size is used. All variables are stored as `double` values. For all measurements the grid is initialized as a square. Thus, the restriction can be done until the coarsest level with a single unknown is reached.

With the included *LAPACK* function [lap99] — described in Section 4.3 — the grid equation can be solved directly. For that the coarsest grid can consist of more than one unknown and the number of grid levels can also be set (with some restrictions) more variably.

In the `Vcycle()` function all necessary parts of the multigrid algorithm are invoked. It starts on the finest level, working recursively until the coarsest grid is reached. In the beginning and in the end of one `Vcycle()` the pre– and post–smoothing steps are done. Therefore the *smooth()* function is called two times for each grid, before the restriction is done and after the prolongation is performed.

When the pres–moothing steps are done the residual is restricted using the `restriction()` function. In order to compute the solution recursively the `Vcycle` function is called on the next coarser grid. Then, on the coarser level the coarse grid representation of the error is solved. If the coarsest grid is reached the solution is computed directly. Depending on the number of unknowns the solution on the coarsest grid can be solved with one iteration using one Gauss–Seidel step. This is the case if the coarsest grid only consists of one unknown. If the coarsest grid contains more unknowns it is computed using a direct solver. For this reason we used the *LAPACK* library to compute a direct solution on the coarsest grid. The LAPACK library is explained in Section 4.3.

As soon as the recursion is done the computed error can be prolongated from the coarser to the finer grid. Therefore the `prolong()` function is called. With this the coarse grid correction is applied to the solution on the next finer grid.

After the V–cycle calls the `smooth()` function, with which the approximate solution additionally is post–smoothed. After that one V–cycle is complete. For our performance measurements we generally used two or four steps each for the pre– and the post smoothing within one V–cycle. The number of V–cycles which have to be executed can be specified with a parameter at the beginning of our multigrid program.

The `restriction()` function is used to solve the coarse grid representation of the error equation. We first compute the residual on the finer level. Then the residual equation which is equivalent with our PDE is solved using its coarse grid representation. The restriction is done using the *full weighting* operator, which we have already mentioned in Chapter 2. The approximation error is initialized with a constant value. The coarse grid coefficients are derived from the corresponding fine grid coefficients, respecting the doubling of the mesh width in each direction.

In the `prolong()` function the error correction is done. The computed error vector is interpolated using the interpolation operator, that is also described in Chapter 2. With this interpolation operator the error correction can be applied to the approximated solution on the finer level. This coarse grid correction reduced the low frequent error parts of the approximate solution on the finer level. When the error correction is done in usually only high oscillating error parts are left in the approximate solution which are eliminated using additional post smoothing steps.

The `smooth()` function is used to smooth the solution of the current grid. Thus every unknown of the grid is passed through using *red–black* Gauss–Seidel iteration. Therefore the grid is labeled in red and black points — as seen in Figure 2.2. The red points are relaxed first and after that all black points are relaxed. In the `smooth()` function also the main optimization methods are done — which are described later in this chapter. The number of smoothing steps which are done for each V-cycle is determined by the number of pre– and post–smoothing steps.

The performance measurement of the program is done counting the execution time and alternative using *PCL* [Ber00]. The PCL library is described in Section 4.4. When the execution time is measured we used the *ANSI C* standard function `clock()`. This function measures the processor time the program needs. This prevents the measure from other influences as there are e.g. I/O operations of the operation system.

Because the program uses `double` values at each point of the grid the floating–point arithmetic of the processor is mainly used. An indication of the number of of floating–point operations that a processor performs is the *MFLOPS* rate. It is defined as:

$$\text{MFLOPS} = \frac{\text{floating point operations}}{\text{seconds} * 10^6} \tag{4.1}$$

For architectures where the number of MFLOPS cannot be profiled directly with PCL, because these CPUs are not supported, the number of floating–point operations, and the number of seconds the program uses is counted. Then the MFLOPS rate can be calculated as shown in Equation 4.1.

All measurements are done from inside the code. With it we only measured the V–cycle algorithm. The initialization functions are not measured with PCL or time calls.

4.3 LAPACK Integration

The *LAPACK* library is a set of mathematical routines, used to solve the most problems which arise for numerical linear problems. The library contains of a set of FORTRAN 77 subroutines that are called inside our multigrid program. The LAPACK library and the corresponding documentation is available at [lap99] where also a more detailed description can be found.

In the multigrid program usually one main problem is at which level the exact solution should be computed. If the residual is restricted until the coarsest grid level is reached then the coarse grid representation consists of only one unknown which can be solved directly. Therefore the LAPACK library is not needed. The equation can be solved using one Gauss–Seidel iteration. For this the grid requires to be a square, which is a limitation for the usage of our program. If non–square grids should be used the coarsest level consists of more than one unknown. Therefore a direct solver is used to compute the exact solution.

Another fact is that when the residual is approximated and restricted to lower grids also the convergence rate is slower as in comparison if the solution is computed directly on the next coarser grid. It is easy to see that if the finest level consists of about one million unknowns and the coarse grid representation only contains one unknown the initial problem might be poorly represented. Therefore also the residual equation and the coarse grid correction are not so accurate if for our example only a few grids are used.

On the other hand a direct solver needs more time to compute the exact solution than if the residual is computed recursively within the multi–level method. In this section we show how a direct solver was implemented with the LAPACK library in our multigrid program and also which advantages and disadvantages arise with the usage of the direct solver.

The direct solution of an equation $A\vec{u} = \vec{f}$ can be done with LU factorization of the matrix A. This requires the following steps:

1. Factorize: $A = LU$ (L: lower triangular matrix, U: upper triangular matrix)
2. Solve: $L\vec{y} = \vec{f}$, with $\vec{y} = U\vec{u}$ (forward substitution)
3. Solve: $U\vec{u} = \vec{y}$ (backward substitution)

If the matrix A is symmetric and positive definite also a special factorization can be done which is called *Cholesky* factorization. This method is better in performance than the general LU factorization because A can be factorized as $A = LL^\top$. Because the LU factorization is out of the scope of our work we only refer to [Sch97].

For our multigrid program we used the general LU factorization. The factorization of the coefficient matrix A needs only be done in the beginning of the multigrid algorithm. Then for every V–cycle backward and forward substitution must be performed on the coarsest level in order to solve the corresponding residual equation directly.

In the LAPACK library the LU factorization routine — which is needed for our multigrid algorithm — is called `dgbtrf`. This function factories a general banded $(m \times n)$ Matrix. Therefore some parameters and arrays must be passed to this function: The number of rows (m) and the number of columns (n) of the matrix A. The number of *superdiagonals* and the number of *subdiagonals* within the band of A. The superdiagonals are the diagonals right from the main–diagonal and the subdiagonals are left. For our multigrid implementation the number of rows and columns correspond with the number of points in x–direction and the number of points to the y–direction. The number of super– and subdiagonals are equal and correlate to the number of points inside a grid line. Next the coefficients matrix A must be transfered in band storage. Also an array for the pivot indices is needed where LAPACK stores which rows of the matrix A were interchanged.

The LAPACK routine `dgbtrs` performs a forward–backward solution step after the matrix A has been factorized using `dgbtrf`. The parameters of the function are equivalent with the parameters described above expect that this function additionally needs the right–hand–side $\vec{f}$.

In difference to the programming language C, the FORTRAN 77 language — in which the LAPACK library is programmed — has certain differences. The first thing is that the function parameters are transfered using *call by reference* in contrast to C, where *call by value* is used.

Another difference is that the arrays in FORTRAN 77 are referenced in another order than in C. In C the fast index is referenced with the last array index and in FORTRAN the fast index is referenced with the first array index. This enforces some array restructuring so that the solution is computed correctly.

4.4 Profiling Tools

In this section we describe the profiling tools that are used for our performance measurements and for a detailed analysis of the code that the compiler produces. The description starts with the *PCL* library. This package provides a common interface to access the performance counters of different architectures. In the next section the *DCPI* system is described with which a detailed analysis of the program binary can be done. The DCPI system consists of a set of various tools which are merely running on the Alpha based architectures.

4.4.1 PCL – Performance Counter Library

The profiling tool which is described here is called *PCL – The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on microprocessors*. The profiling tool is developed from *Rudolf Berrendorf* at *Forschungszentrum Juelich GmbH*, Federal Republic of Germany. The tool is provided for free use and can be found at [Ber00].

PCL mainly concentrates on events that are influenced from the memory hierarchy of the computer architecture. For this reason the PCL tool is used to measure various events that give information about the performance behavior of our multigrid program. Unfortunately the PCL tool did not support the Athlon and the HP PA8500 processors, hence with this architectures only time measurements could be done.

For our measurements we used version 1.3 in the beginning. With the release of version 2.0 in September 2000, we switched to the new version.

The PCL tool provides access to the *performance counters* that are part of the microprocessors. A performance counter can measure performance–relevant events for different microprocessors. The number of performance counters a processor contains differs between the existing processors. Also the types of events that can be measured with them. The reason for this is that the manufactures have different views which events are important for performance analysis. Moreover performance counters increase the costs of the CPUs.

PCL supports – depending on the processor type — a various set of event types. The number of events that can be measured in simultaneously also differs for each CPU and depends on the number of performance counters that are included in the processor.

In the following different event types that are supported with PCL are shown in Figure 4.1. The figure also shows which of these events are available on the processors which are used within this thesis.

The different events which can be measured with PCL can be split into three parts: The events that concern the memory hierarchy. These are the L1 and L2 cache misses and the translation lockaside buffer (TLB) misses. The events that concern the instruction categories are the numbers of cycles, floating and integer instructions, load/store instructions, jumps and the atomic operations. The last category are the functional stalls. In our multigrid program we mainly measured the L1 and L2 cache misses and the MFLOPS rate. The MFLOPS rate that is calculated within PCL is measured similar to equation (4.1):

$$\text{MFLOPS} = \frac{\text{PCL\_FP\_INSTR}}{\text{PCL\_CYCLES}} * \text{MHz rate} \tag{4.2}$$

The PCL tool can be used for external and internal measurements. As we claimed above all measurements are done inside our program. The following PCL functions are used:

Events	Pentium II	21164	21264	Hitachi SR8000
PCL_L1ICACHE_READ	⊗	-	-	-
PCL_L1ICACHE_MISS	⊗	⊗	-	⊗
PCL_L1ICACHE_HIT	-	⊗	-	-
PCL_L1DCACHE_READWRITE	-	⊗	-	-
PCL_L1DCACHE_MISS	⊗	-	-	⊗
PCL_L1DCACHE_MISSRATE	⊗	⊗	-	-
PCL_L2CACHE_READ	-	⊗	-	-
PCL_L2CACHE_WRITE	-	⊗	-	-
PCL_L2CACHE_READWRITE	⊗	⊗	-	-
PCL_L2CACHE_HIT	-	⊗	-	-
PCL_L2CACHE_MISS	⊗	⊗	-	-
PCL_L2DCACHE_READ	⊗	-	-	-
PCL_L2DCACHE_WRITE	⊗	-	-	-
PCL_L2DCACHE_READWRITE	⊗	-	-	-
PCL_L2DCACHE_MISSRATE	-	⊗	-	-
PCL_ITLB_MISS	⊗	⊗	⊗	⊗
PCL_DTLB_MISS	-	⊗	-	⊗
PCL_CYCLES	⊗	⊗	⊗	⊗
PCL_ELAPSED_CYCLES	⊗	⊗	⊗	-
PCL_INTEGER_INSTR	-	⊗	-	-
PCL_FP_INSTR	⊗	⊗	-	⊗
PCL_LOAD_INSTR	⊗	⊗	-	⊗
PCL_STORE_INSTR	-	⊗	-	⊗
PCL_INSTR	⊗	⊗	⊗	⊗
PCL_JUMP_SUCCESS	⊗	-	-	-
PCL_JUMP_UNSUCCESS	⊗	⊗	-	-
PCL_JUMP	⊗	-	⊗	-
PCL_STALL	⊗	-	-	-
PCL_ATOMIC_SUCCESS	-	⊗	-	-
PCL_MFLOPS	-	⊗	-	-
PCL_IPC	⊗	⊗	-	-

Figure 4.1: PCL: Supported events with PCL and their availability on different architectures.

- `PCLquery()`: Query if the events that should be measured are available for the CPU.
- `PCLinit()`: Initialize a thread specific descriptor. PCL since version 2.0 needs this in order to make the event information more thread safe.
- `PCLstart()`: Start the PCL measurement. The events are passed with this function.
- `PCLstop()`: Stop counting the specified event types. The results of the measurements are written in to a list of variables.

4.4.2 DCPI: Compaq (Digital) Continuous Profiling Infrastructure

In this section an overview of the *Compaq* (formerly Digital) *Continuous Profiling Infrastructure (DCPI)* is given. The DCPI system consists of a set of tools. With these tools a low–overhead continuous profiling of the executables can be done. The DCPI system — similarly to the PCL tool — also reads the performance counters that are integrated in the processors. However, in contrast to PCL, the DCPI system is only running on the Alpha systems.

The profiling is done using a periodic *sampling* for the performance counters. The sampling rate can be influenced with the *sampling period*. This describes how often an event can occur until an exception is made, because it does not make sense to make an exception each time the event appears. This would seriously influence the profiling results or even make code execution impossible.

Since the DCPI system comprises a lot of tools only the important ones for this thesis are described in the following. For a complete overview one can e.g. take a look at [Com00].

Every time program binaries should be profiled with DCPI, a daemon (`dcpid`) must be started before the program is executed. With this daemon the event types that should be measured can be specified. Depending on the type of the event and the number of performance counters of the CPU, different events can be measured in parallel.

The `dcpid` tool writes an image file for each program binary that is profiled. When the program binary stops the daemon can be halted with the `dcpiquit` program. With this information the DCPI profiling tools can be used to investigate the output that `dcpid` has made.

The mostly important DCPI tools that are used within this thesis are:

- `dcpiprof`: This tools shows the number of samples spent for every function according to the selected event. Inside the output all function names are listed with the according number of events used and their number of samples measured. With this one can gain an overview in which functions the most samples are used. This function can be compared with the `gprof` [gpr98] command which produces a similar output.
- `dcpilist`: In comparison to `dcpiprof` a procedure listing with the profiling information inside a function can be given. It shows the number of samples that are spent for each event for execution an instruction in the specified function. Also the machine code which is generated from the source code by the compiler can be listed. With this information good conclusion can be done, e.g. in what way the compiler can optimize the source code. Also the source code lines which produce bottlenecks or performance loss can be indicated.
- `dcpiwhatcg`: The `dcpiwhatcg` tool lists the number of cycles that were used for executing all instructions of the program compared with the number of stalls. The stalls can be of various categories. With this tool one can see for which instructions the most cycles are needed. Additionally in combination with the `dcpilist` tool also the exact location inside the source code can be found where the instruction or the stall is produced.
- `dcpicalc`: The `dcpicalc` tool can be used to show how many cycles are spent for executing the program and for what purpose they are used. In comparison to `dcpiwhatcg`, this tool provides an overview of the *dynamic* and the *static* stall cycles that are used for the program. It also can give the causes of these stalls.

The `dcpicalc` manual identifies static stalls as these stalls that even occur in the best case for an instruction. This means e.g. that if a memory load hits the cache it also takes time until the information is available. The other stalls that occur are labeled dynamic. If a program is optimized also the optimized version should have fewer dynamic stalls than the unoptimized version. If for a theoretic example a program produces only static stalls it can be seen as optimal.

The previously described profiling tools should give a short overview of the DCPI system. On the basis of the tools that we have mentioned a profiling example example is given at the end of Chapter 5.

4.5 Program Profiling

When a program is to be optimized it is helpful — and strongly recommended — to use several profiling tools in order to show where most of the execution time is spent. Because it does not make sense to speed up program parts which are rarely executed whereas other parts consume the major part of the execution time and are poorly implemented.

This is a conclusion from *Amdahl's Law.* It states that if a subroutine which uses $n\%$ of execution time is optimized, the whole program can at most be accelerated by the factor:

$$\mathrm{S} = \frac{1}{1 - \frac{n}{100}} \tag{4.3}$$

The frequent parts of a program — the program parts where most of the cycles are spent — should be speed up rather than the infrequent ones, where much fewer cycles are spent.

To make these dominating parts recognizable a profiling tool is used to show which routine consumes the most execution time. For the present multigrid algorithm we used the `gprof` utility. The `gprof` tool is available on various architectures. As described above the `gprof` command is similar to the `dcpiprof` tool. A complete documentation can be found at [gpr98].

The multigrid program therefore is started on a grid with about one million unknowns. For that we used an interval size of 1024 in each direction. The red–black Gauss–Seidel smoother is executed four times as pre– and post–smoother each. The program was profiled using the `gprof` tool with 10 V–cycles. The resulting output is shown in Table 4.2.

In the table we can see that the `smooth()` function takes about 85% of the time of the whole multigrid program. Therefore we can determine the `smooth()` function as the dominating part of our multigrid program. This results does not astonish, because the algorithm is based on relaxing each unknown multiple times using the method of Gauss–Seidel.

Furthermore in the fourth column (*calls*) the total number of calls to the `smooth()` function is given.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
84.18	41.04	41.04	180	228.00	228.00	smooth
8.31	45.09	4.05	90	45.00	45.00	restriction
6.22	48.12	3.03	90	33.67	33.67	prolong
1.29	48.75	0.63	1	630.00	630.00	init_grid
0.00	48.75	0.00	10	0.00	4812.00	vcycle
0.00	48.75	0.00	1	0.00	0.00	define_variables
0.00	48.75	0.00	1	0.00	0.00	free_grid
0.00	48.75	0.00	1	0.00	0.00	generate_grid
0.00	48.75	0.00	1	0.00	0.00	getOptions

Figure 4.2: Profiling of the 2–D multigrid algorithm using `gprof`

To demonstrate that the output of the `grpof` tool is correct we counted the number of smooth calls by hand. For our interval size of 1024 in each direction we have 10 levels. At nine levels the smooth function is called. The coarsest level is solved directly with one iteration, but not using the `smooth` function. For this we have nine levels where at each level the pre– and the post–smoothing steps are done. This leads to 18 `smooth` function calls. Due to the fact that we did 10 V–cycles the `smooth` functions calls increases up to 180 calls. This is what can be found in the fourth column for the `gprof` output.

Also the `restriction()` and the `prolong()` functions are shown. These function both consume about 15% of the total execution time. The other functions were only executed once and have the purpose to initialize the grid and the variables at the beginning of the multigrid iteration process.

The conclusion from the `gprof` command leads to the result that the major optimizations have to address the `smooth()` function. In the `restriction()` and the `prolongation()` functions only about 15% of the execution time are spent. These parts can be labeled as the less infrequent time–consuming parts of the multigrid program.

4.6 Program Parameters

The implementation of the multigrid program has different parameters which determine the grid size, the number of pre– and post–smoothing steps, different events for PCL, the number of V–cycles that should be performed, etc. These parameters can be found in Appendix A.

All optimizations which are applied to our multigrid implementation are stored inside the program. For the reason — shown in Table 4.2 — we concentrate on the `smooth()` function. For this we have applied multiple optimizations methods for the `smooth()` function to increase the performance of the whole program.

This gives us the possibility to compare the performance of these different versions. This also leads to a better reproduction when maybe further optimization methods are applied to the program. The different versions for all optimizations — which are described in the following chapter — are declared in Appendix B.

Chapter 5

Cache Optimizations

This chapter describes the different optimization methods which have gradually been integrated in our 2–D multigrid code. The chapter starts with the *loop interchange* optimization which is easy to implement, but nevertheless can lead to impressive performance gain. For this two locality concepts need to be explained. The both concepts are labeled temporal locality and spatial locality. Both refer to a word inside a block which is fetched from main memory to a cache line. Temporal locality means that this word is referred to in the near future. If another word of this block is also relevant and referred to we call this spatial locality.

Furthermore the optimization methods continue with the realization of different data layouts, loop fusion and blocking methods. In the end of the chapter we have been made a profiling experiment in order to show how the compiler benefits form the implemented optimizations and furthermore to explain the methods that the compiler has been used to improve performance. For all optimization methods performance measurements are done in order to show the speed increase and the cache efficiency that is obtained.

All experiments are done one several architectures, which are described in Chapter 3. The experiments are measured at least four times to avoid other influences. The parameters that were used for our program have been varied and are described whenever a new optimization method is explained. An overview of the used cache optimization methods and techniques to improve the cache efficiency of iterative methods can be found at [DS99, Han98, Los98, Rüd97, WKKR99]

5.1 Loop Interchange

In this section we take a look at the optimization method called *loop interchange*. In our multigrid program one 2–D data array is accessed to update all grid points within the Gauss–Seidel iteration. Therefore two loops — one for each dimension — are used. This is called a *loop nest* because one or more loops are embedded within other loops. In order to access this array without having a loss in performance due to occurring cache misses, the loops should be ordered with respect of how the program accesses the array elements. The best way to gain such an optimization is to access all used data in sequential order.

This data access method is called a *unit stride*. Because of the spatial and temporal locality of the resulting references, the data can be accessed very fast, because it is still localized in the caches leading to high cache reuse and reducing the number of cache misses. In comparison to the possibility that all data of the array are referenced in non–sequential order this leads also to a more efficient usage of the cache levels.

In order to demonstrate this effect we consider the following 2–D array: `a[y][x]`. The addressing mode for the C programming language lets the last index run fastest, which means that `a[y][x]` and `a[y][x+1]` are stored adjacently in memory. To use the caches effectively the outer loops should include the y–statement, because neighboring values in this dimension are farther away from each other in memory — depending on the size and the number of x–values.

The inner loop now contains the fast running index which is the x–value in this case:

```
for (y=1; y <= max_y; y++){
   for (x=1; x <= max_x; x++)
      a[y][x] = ...;
}
```

This optimization method is easy to implement. Also *loop unrolling* optimization which is done by the compiler is more efficiently for unit strides. If the compiler detects that multiple accesses to the array `a[y][x]` are made the inner loop can be unrolled if no data dependencies are violated.

The following demonstrates a unit stride with two operations unrolled (with `max_x` ≥ 2, and `max_x` is even to beware all data dependencies):

```
for (y=1; y <= max_y; y++){
   for (x=1; x <= max_x; x=x+2){
      a[y][x]   = ... ;
      a[y][x+1] = ...;
}  }
```

If the inner loop is very small it is also conceivable to unroll the outer loop instead of the inner. For small arrays the cache effect is not so high than for an array which consists of several thousands of elements — also depending on the cache size of the used architecture. For small arrays it is possible that all data can be held in the faster caches and for that optimization is not strongly recommended. However small data sets do not play an important role in high performance computing because usually the amount of data is much larger to fit in the caches of the CPUs.

The previous example of loop fusion optimizations refers to our 2–D multigrid code. We also have 2–D arrays where all grid points are stored. Neighboring points to the left and the right (for one grid line) are stored adjacently in memory. The upper and lower points are located farther away in memory — depending on the number of points in one grid line. In order to demonstrate the resulting speed increase we have measured the CPU time in seconds which is spent for our 2–D code using non–unit stride and unit stride loops. The results of this measurement can be seen in Figure 5.1.

With the results of Figure 5.1 one can see that the CPU time for the Alpha 21164 even halves when unit stride references are made in comparison to the non–unit stride cases. The CPU time improves from about 88s to 40s. For this example about 83 millions of floating–point operations are made within one V–cycle. As there are 10 V–cycles the number of floating–point operations scales with

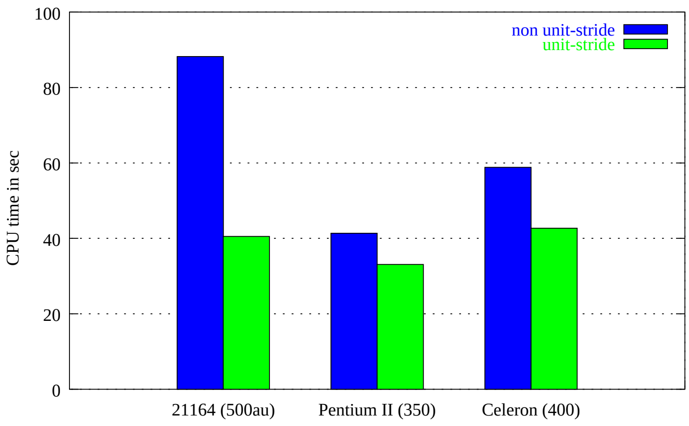


Figure 5.1: Comparison of used CPU time between non–unit stride and unit stride array accesses for a grid containing about 1 million unknowns with 10 V–cyles.

the same factor to 830 million operations. This leads to a performance of about 10 (!) MFLOPS[1] with non–unit stride and 20 MFLOPS for the unit stride loop for the Alpha 21164 architecture.

The speed increase for both Intel architectures is much smaller but also the non–unit stride is faster. To gain more insight we investigated the number of L2 cache misses for this example.

In Figure 5.2 the impact of the cache misses is shown for the Alpha 21164 and the both Intel architectures. The L2 cache misses are measured with the same multigrid problem as described above, with the limitation that only one V–cycle is done. These results also show the influence the cache sizes with their different implementation methods have on the performance. For the Alpha 21164 the number of L2 cache misses is almost halved. In comparison the L2 cache misses that occur for the Intel architectures are much smaller for non–unit stride accesses. The L2 cache misses for the Pentium II also could not be improved highly. Only for the Celeron CPU lower L2 cache misses can be found.

This reason for the different effects can be explained with the following facts:

1. The Pentium II has a larger L2 cache than the other architectures compared. It has 256 KB in comparison to 128 KB for the Celeron and 96 KB for the Alpha 21164.

2. The cache levels of the Pentium II are arranged differently than those of the Alpha 21164 architecture.

For the Alpha 21164 we must also note that it has three cache levels. The L3 cache is 4 MB and direct mapped while the L2 cache is only 96 KB in size. For this the comparison between the 512 KB cache of the Pentium II must be seen with care. Unfortunately the L3 cache misses could not be measured with PCL. But the measured performance results also lead to the assumption that the direct mapped L3 cache is also more sensitive for non–unit stride accesses than the both Intel architectures which can lead to enormous performance varieties.

Because the Celeron CPU has the smaller L2 cache size in comparison to the Pentium II it suffered from more L2 cache misses for non–unit stride accesses. Because the Celeron has the same cache strategy (set associativity, etc.) than the Pentium II the difference for the L2 cache misses was not so high in comparison to the Alpha 21164. It should also be distinguished that the presented optimization is easy to implement for this example but also easy to overlook. In the previous example, the optimization methods leads to roughly 100% performance gain for the Alpha 21164.

The compiler can generally not detect if there are data dependencies between these loops, and if a rearrangement could lead to different computation results. In the previously described examples for the C programming language it is easy to see that there are no dependencies between the arrays. Due to the fact that for a compiler an array behaves like a pointer, it cannot determine if multiple

[1]The Alpha 21164 (500au) provides a peek performance of 1000 MFLOPS.

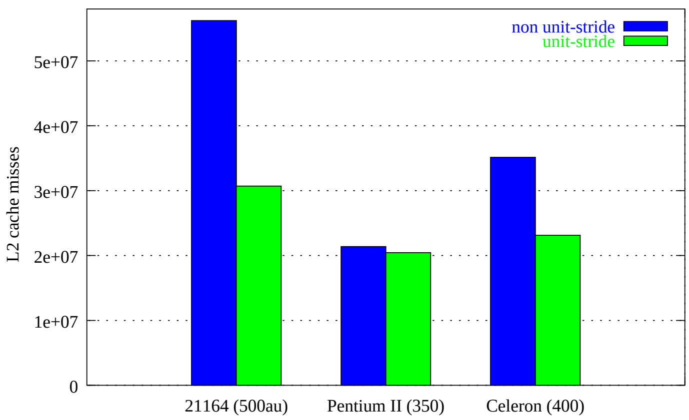


Figure 5.2: L2 cache misses for non–unit stride and unit stride data accesses.

arrays are linked together. Hence, in general, it cannot ensure that no data dependencies occur when a loop interchange is performed. Thus, in many cases, it is the task of a programmer to implement this optimization method.

5.2 Datalayout

In this section different techniques to obtain better data locality with data layout methods are described. In the beginning several data structures are created to show the influence that different storage schemes can have with respect to data locality aspects. With the resulting data layouts the influence of *array padding* is examined. With it occurring thrashing effects can be eliminated. Also the method of array merging is presented within this section.

5.2.1 Data Structures

We have already mentioned in Section 4.4 that the `smooth()` function consumes most of the execution time. In this function every unknown in the grid is relaxed with the Gauss–Seidel algorithm to compute a new approximation for the solution.

Before we have implemented different data structures we investigated the linear systems under consideration:

$$A_h \vec{u}_h = \vec{f}_h \qquad \text{with } A_h \in \mathbb{R}^{n \times n}, \vec{u}_h, \vec{f}_h \in \mathbb{R}^n \tag{5.1}$$

The variable n denotes the number of unknowns in each dimension. The matrix A_h contains the variable coefficients of the 5–point–stencils. Thus for each matrix rows coefficients for the accesses to the west, east, north, south and center direction have to be stored.

If e.g. a 1024x1024 grid is used there are about one million unknowns. For this example the matrix A_h consists of about 5 million coefficients because for every unknown five coefficients have to be stored.

Both vectors $\vec{u}_h$ and $\vec{f}_h$ contain around 1 million values each. Because all of these values are common 8 bytes per double precision number this leads to a data size about 56 MB. This memory requirement is too large for the architectures used in this thesis to completely fit into the cache levels. For this reason it is important to implement a data layout that has the benefit of good data locality which helps to avoid lots of cache misses before other optimization methods are applied.

In the following eight different data structures are discussed which are implemented in our 2–D multigrid program. The data structures can be found starting at Figure (5.3) to (5.10).

- **Data Structures 1+2:** The first two data structures are implemented using one array which holds all values. The data layout for data structure 1 is shown in Figure 5.3 and data structure 2 is shown in Figure 5.4. Inside the array for data structure 1 all values of the vector $\vec{u}_h$ and the entries of Matrix A_h are stored first. Each row of A with the corresponding unknown value is stored, followed by the next matrix row. At the end of the array all entries of vector $\vec{f}_h$ are stored. In data structure 2 the line entry of the right–hand side $\vec{f}_h$ is also stored with the coefficients and the unknowns of $\vec{u}_h$ for the same row/equation.

- **Data Structures 3+7+8:** First all values of $\vec{u}_h$ and then all values of $\vec{f}_h$ are stored. Behind them the coefficients of A are stored in bandwise manner. In order to show the influence of different *array padding methods* [TCM94, TR98] we implemented data structure 7 and data structure 8 which use different padding blocks. In comparison to the other data layouts for for data structure 7 we allocated an array for each band.

- **Data Structure 4:** The whole solution vector $\vec{u}_h$ is stored first. Then the right–hand side $\vec{f}_h$ and all coefficients are merged together.

- **Data Structure 5:** The solution $\vec{u}_h$ is merged with $\vec{f}_h$. Then all coefficients which are also merged for each matrix row are stored.
- **Data Structure 6:** In this data structure $\vec{u}_h$ and $\vec{f}_h$ are stored after each other. Furthermore the coefficients are merged and inserted at the end of the array.

For each data structure we marked with a shaded triangle where padding values are inserted. For triangles with different colors different padding block sizes are inserted. Usually the padding data are inserted at the end of a grid line. Only for data structure 3, 7 and 8 the padding values differ. This will be investigated in Section 5.2.2.

For all data structures we have made made measurements to show their impact on the performance. The measurements were done using 1024 intervals in each direction, 10 V-cycles and two pre– and two post–smoothing steps with our 2–D multigrid code. Figure 5.11 shows the measured for the CPU time in seconds on different architectures. Due to the different rating which the architectures have in comparison to each other, also the execution time between the data structures differs from each other on the same CPU.

To investigate an efficient data structure it is particularly important to examine the performance of the `smooth()` function — that we already showed in Section (2.9) — which is done for every unknown of the grid to gain a new approximation.

For each unknown of the 2–D grid, with the use of a 5–point–stencil, the following information is needed:

$$u_{ce}, u_{no}, u_{so}, u_{we}, u_{ea}, a_{ce}, a_{no}, a_{so}, a_{we}, a_{ea}, f_{ce}.$$

The references that one unknown has to make for the Gauss–Seidel method can be seen in Figure (5.12).

For data structure 1 and data structure 2 every grid point is stored with its according coefficients and the right–hand side. This means that for every reference to the neighboring point its according data additionally is brought in the cache which is currently not needed. Because this data is not needed it cannot be reused, but when the upper point is accessed this data is also loaded in the cache. With such a cache load also other cache lines have to be replaced. If a cache line which is replaced, contains information about points in the same line which are relaxed next, the cache reuse is very low leading to an increase of cache misses. These cache misses increase the execution time, because load and store instruction, need a lot of cycles. Also other operations which need data that is replaced from the cache have to wait, which leads to an increasing number of dynamic stalls.

As shown in Figure 5.11 data structure 1 and data structure 2 have slower performance than for example data structure 4. Between the other data layouts no significant difference can be seen. Therefore we additionally investigated the L2 cache misses. In Figure 5.13 an overview of the cache misses which occur using all data structures is given. There for the performance measurements we have used the same parameters as used for the execution time experiments, with the restriction that only one V–cycle has been performed. One can see that the cache misses for data structure 4 are lower than for data structure 1 and 2 leading to a faster execution time. Also the L2 cache misses for data structure 4 are lower than for all other data structures. The detailed analysis of these results follows further in Section 5.2.2.

So far, we have implemented different data layouts. However, with the measured results no valid statements can be made due to possible trashing effects, which are explained in detail in Sec-

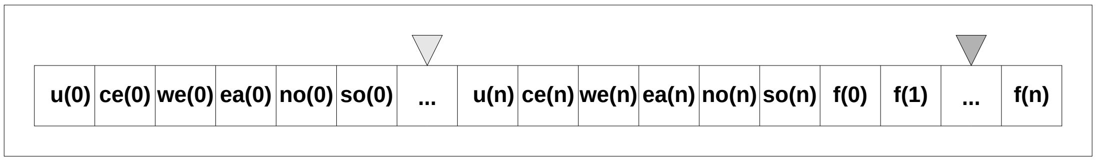


Figure 5.3: Data Structure 1

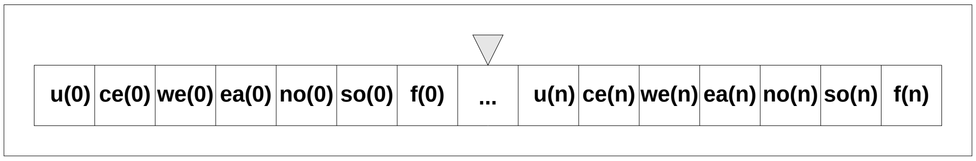


Figure 5.4: Data Structure 2

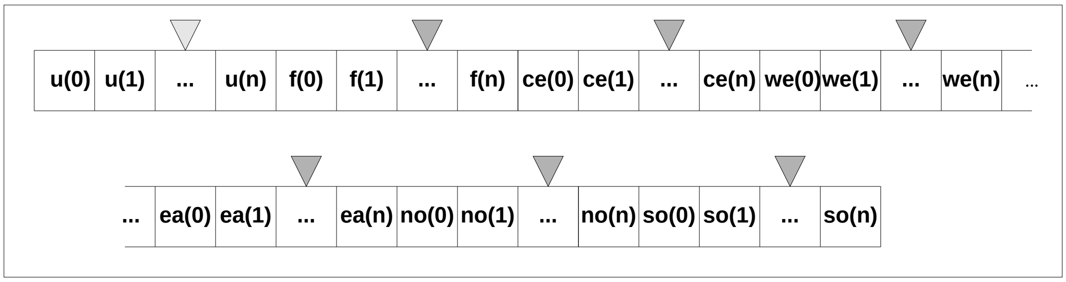


Figure 5.5: Data Structure 3

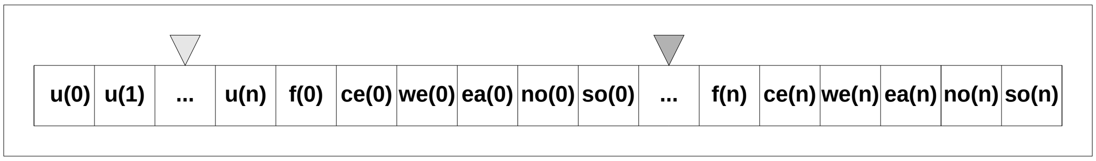


Figure 5.6: Data Structure 4

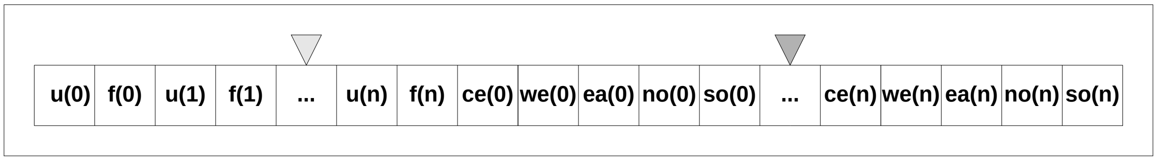


Figure 5.7: Data Structure 5

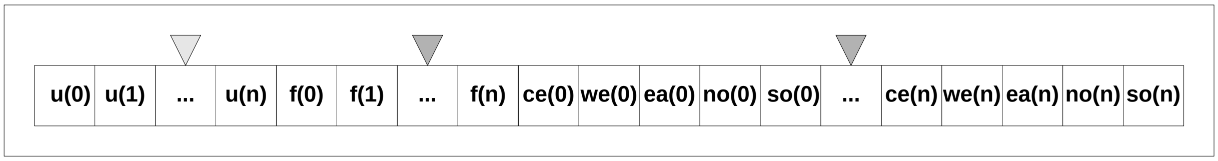


Figure 5.8: Data Structure 6

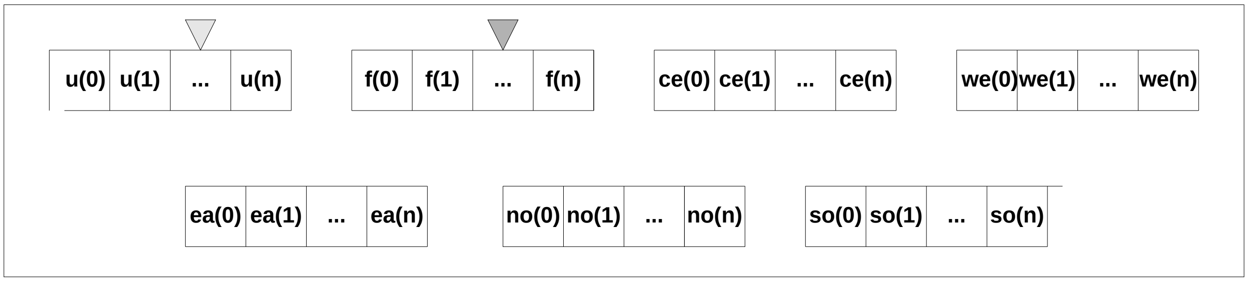


Figure 5.9: Data Structure 7

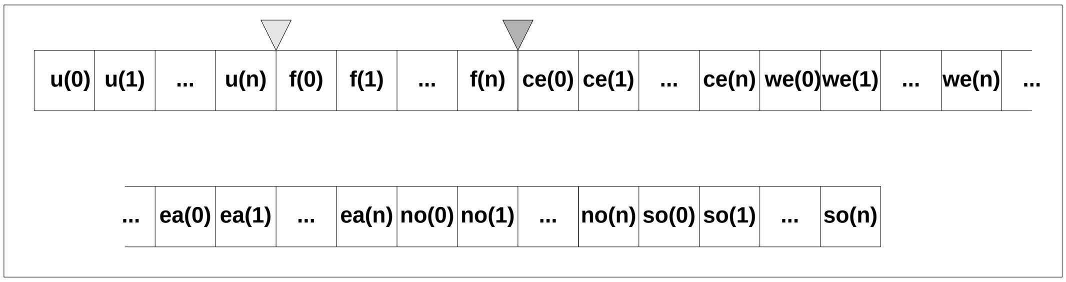


Figure 5.10: Data Structure 8

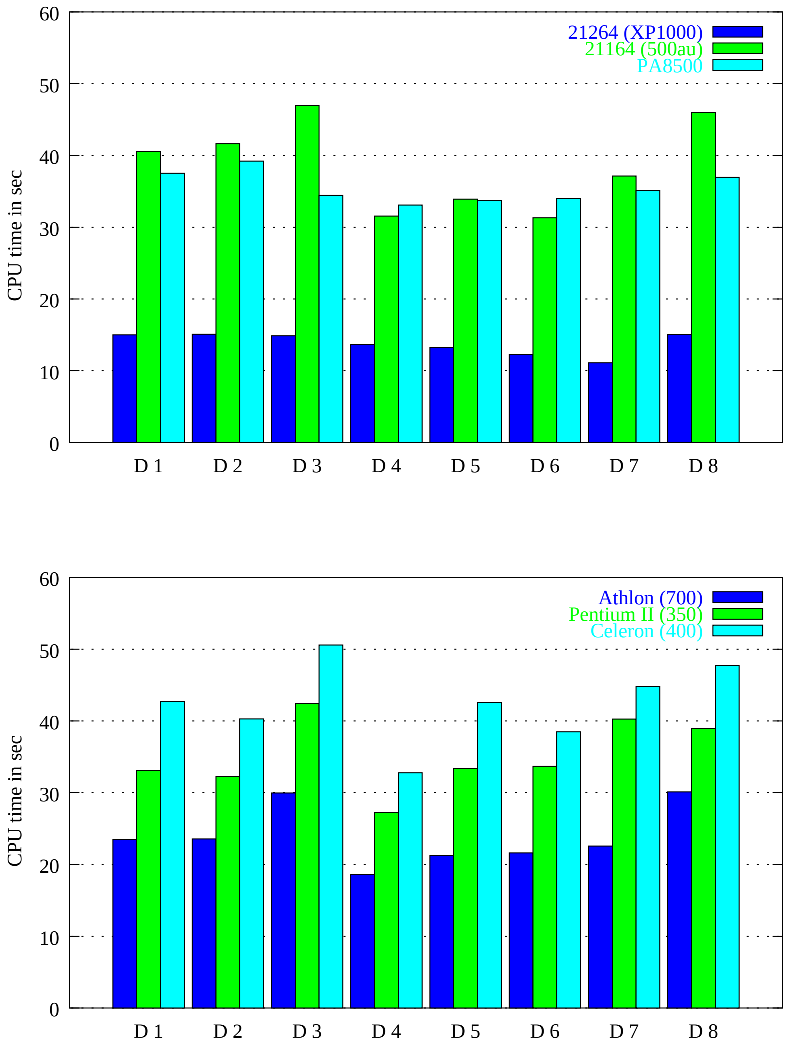


Figure 5.11: Comparison of the CPU times for different data layouts.

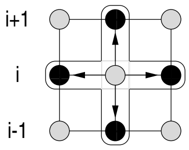


Figure 5.12: Relaxation of a red unknown with a red–black Gauss–Seidel algorithm.

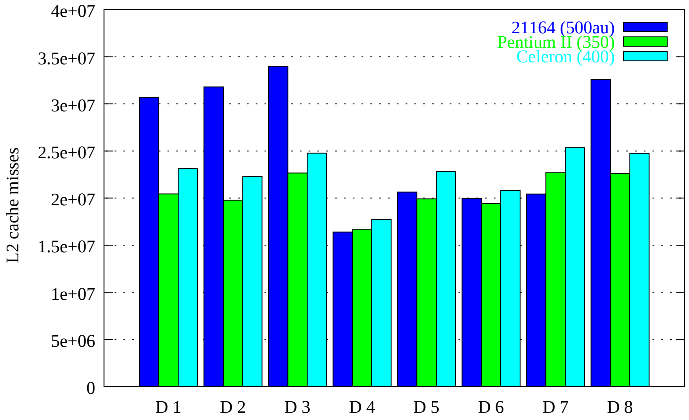


Figure 5.13: Overview of the L2 cache misses over all data structures.

tion 5.2.2. We only concentrate on the implementation of data layouts that should lead to a more efficient cache reuse with the observation of the Gauss–Seidel method and the amount of data that is needed for our variable coefficient problem. The only fact that we have already noted is that for data structure 4 the number of cache misses significantly has reduced, compared with the other data layouts.

5.2.2 Array Padding

In this section we investigated the influence of array padding. The goal of array padding is usually to eliminate *thrashing effects*. These effects can occur if multiple cache lines frequently replace each other, because data from both memory blocks are accessed. With the additional data which is inserted into these arrays, they are stretched, forcing them to be mapped to different cache lines. The result is that both memory blocks do not replace each other any more, leading to more efficient cache usage and decreasing cache misses.

For the array padding optimizations we started with data structure 3, data structure 7 and data structure 8. For these data layouts the data is stored in bandwise manner. Because all values of the solution vector $\vec{u}$ are stored one after the other — in comparison to data structure 1 and data structure 2 — the coefficients do not need to be loaded if a value from $\vec{u}$ is accessed and brought to the cache. With a reference to a component of the solution vector $\vec{u}$ also its neighboring value to the west and east which reside in the same memory block are loaded to the same cache line. As we have shown in Figure 2.9 the information about its neighboring points is needed for the grid point which is to be relaxed. With this the cache should be reused in a more efficient way.

Nevertheless the execution time is higher than for data structure 1 and 2. An analysis of the L2 cache misses in Figure 5.13 shows that more cache misses occur. The cause for this effect are thrashing effects. In Figure 5.12 we have described in which way each point is updated in a Gauss–Seidel iteration. Because all unknown grid points are stored in sequential order the upper and lower points — which are also accessed — are likely to not reside in the same memory block[2], which is already in the cache. The upper and lower points which are labeled as u_{no} and u_{so} reside in memory locations which are farther away. If these memory blocks in which the upper and lower points reside probably are mapped to the same cache line where also u_{ce}, u_{we} and u_{ea} are mapped this results in serious thrashing effects. Every access to the upper and lower points brings along several cache misses, because almost all needed values are mapped to the same cache line — nevertheless residing in different memory blocks.

Figure 5.14 shows an example thrashing for a direct mapped cache. The point in the upper line of the grid $u_{no} = u(1, i+1)$ is mapped the same cache line as the center point $u_{ce} = u(1, i)$. However, they are located in different memory blocks. An access from u_{ce} to u_{no} forces the cache lines to be

[2]depending on the grid line size in x–direction

replaced. This produces unnecessary cache misses because the caching strategy fails. This effect repeats for all further iterations intensifying the performance decrease.

We have implemented two possibilities to prevent thrashing effects. The first method is to use array padding. The other method is *array merging* which is described in Section 5.2.3.

The array padding method inserts data blocks in the memory blocks so that the data parts are not mapped to the same cache lines any more. The data layout is changed in such a way that leads to a better spatial cache reuse. This is the main goal of array padding. Memory blocks which are mapped to the same cache lines are stretched so that the involved data are brought to other memory blocks and therefore to other cache lines, because the array is filled. With this also the data is loaded in another cache line producing fewer thrashing.

Array padding can be done in two different ways [TR98]; the following examples are shown in FORTRAN 77 notation:

- **inter padding**: Padding data is inserted between arrays.
  Example:

  ```
  double precision a(N), b(N) -> a(N), BLOCK(padsize), b(N)
  ```

  In the FORTRAN 77 programming language the code transformation above would lead to a changed base address for array `b(N)`[3].

- **intra padding**: Padding data is inserted within an array.
  Example:

  ```
  double precision a(N,N), b(N,N) -> a(N+padsize, N), b(N+padsize, N)
  ```

Note that for the previous examples the dimension size of the arrays has changed.

For our 2–D multigrid program only intra padding is used. We investigated in detail intra padding optimizations for data structure 3, data structure 7 and data structure 8. The storage schemes of the three data structures do not differ significantly, but we used different padding sizes for each data layout. For this we can show the performance of the different padding variants in more detail.

In contrast to FORTRAN 77 in the C programming language two allocated arrays usually are not allocated directly behind in memory. This fact usually makes intra padding useless in C, because the distance for the both arrays is random. For the previous example `a(N)` and `b(N)` usually are

[3] In difference to the C programming language, in FORTRAN 77 the arrays which are allocated directly behind each other are also located behind each other in memory.

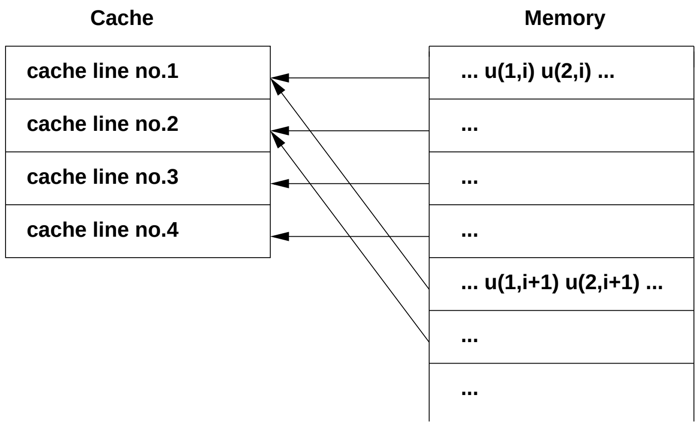


Figure 5.14: An example of *thrashing effects* (FORTRAN 77 notation): multiple memory blocks are mapped to the same cache line.

not located one after the other in memory in C. For our data layouts we used intra padding with the intention to insert padding blocks of different sizes within the band or between different bands. With this the memory distance between neighboring points in different grid lines gets larger, yielding that both memory blocks are mapped to different cache lines.

The result for this optimization is shown in Figure 5.15. In this figure the best L2 cache miss rate for a set of padding variables can be found. For this the three data structures are compared with the unoptimized data layout, where no padding values are used. There it can be seen immediately that the Alpha 21164 CPU benefits significantly from the previous optimization. Although we have not investigated the L3 cache misses for the Alpha 21164 we suppose that the cache usage suffers from thrashing. The reason for this is that the L3 cache is implemented as a direct mapped cache.

For data structure 3 intra padding is used. Inside $\vec{u}_h$, $\vec{f}_h$ and the coefficients additional blocks are inserted. To verify in detail which conflicts between or inside the bands occur we used data structure 7 (intra padding) and data structure 8 (simulated inter padding) to gain a more detailed view. The measurements provide the following results:

- For data structure 3 it is investigated that padding inside the values of $\vec{u}_h$ (intra padding) almost has no effect on the cache misses and the execution time. The best results where observed if the values of $\vec{f}_h$ are padded. This leads to the presumption that values for the different bands possibly used the same cache line and for that have conflicts. On the other hand the fact that padding inside $\vec{u}_h$ almost has no effect on the execution time possibly signifies that these values do not have conflicts which each other. We exclude the possibility that the values of $\vec{f}_h$ suffer from conflicts because these values are used only once in each iteration step. The same proposition effects the bands for the coefficients, where also conflicts between other bands are most probably. To verify the presumption that there are no conflicts inside the bands but between the different bands the both additional data structures are examined.

- To measure only the intra padding effects we used data structure 7. From this it follows that this data layout did not significantly profit from padding. This effect supports the assumption we made before that the different bands are possibly in conflict with each other and not the values inside. Because all bands are stored in different arrays this can also be seen as a kind of random inter padding. The space between the different arrays cannot be determined because it is done flexibly by the compiler. Hence, this measurement cannot only show the effects of intra padding for the data layout, because of these inter padding effects. Therefore we implemented data structure 8 to show only the inter padding effects.

- In data structure 8 the values are stored in the same way as in data structure 3, with a different padding method applied. We only inserted padding blocks between the bands and

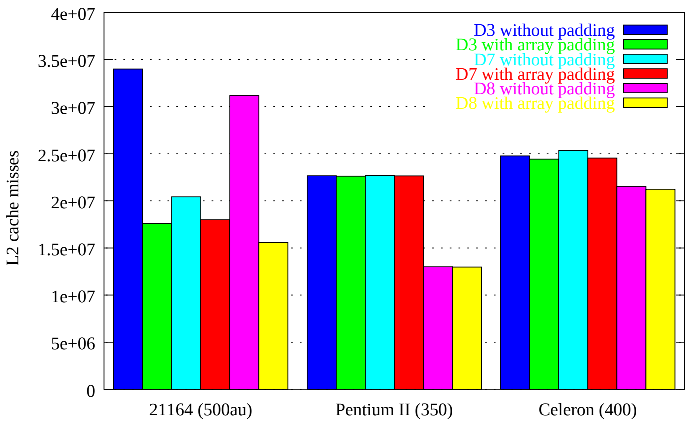


Figure 5.15: Overview of the L2 cache misses that occur with and without array padding for the selected data structures.

not inside. This method can also be seen as a kind of inter padding (with no intra padding). The performance increased if padding blocks are inserted between the different bands. For the reason that the execution time almost is the same as in data structure 3 this leads to the assumption that only the different bands have conflicts. This also validates the proposition that was made in data structure 3.

In the Figures 5.16, 5.17 and 5.18 we show the measurement results for a padding experiment to make these effects visual. For a grid with 1024 intervals in each dimension, 10 V–cycles and two pre– and post–smoothing steps with the method of Gauss–Seidel are applied. All measurements are done using data structures 3, 7 and 8. The padding values to the x–direction note the value for the first triangular for each data structure. The padding values to the y–direction mark the second triangular values. The position of the shaded triangulars for each data structure can be found in Section 5.2.

In Figure 5.16 one can see that for the Alpha 21164 CPU the MFLOPS rate does not increase for ascending values to the x–direction. The padding values on the x–axis — labeled as `pad[x]` — describe the intra padding values for the solution vector $\vec{u}_h$. Because the MFLOPS rate does not increase for different padding blocks, the values of $\vec{u}_h$ have no conflicts. For raising values to the y–direction the MFLOPS rate increases. The padding values form `pad[y]` describe the intra padding between the other bands behind $\vec{u}_h$ which are $\vec{f}_h$ and the coefficients. This supports the assumption which is made before. The solution vector $\vec{u}_h$ possibly suffered from conflicts with other bands or maybe the bands of the coefficients or the right–hand side vector $\vec{f}_h$ have conflicts. For the Alpha 21264 CPU the MFLOPS rate has the same behavior for the padding values in x–direction. For padding values in the y–direction the MFLOPS rate is more sensitive if very small values are used. For small ascending values there are some lines where the performance drops. There the padding values still are set too small to prevent thrashing. This lack of performance disappears if `pad[y]` is chosen large enough. The performance regression indicates that for small values still conflicts occur which vanish with ascending values to the y–direction. Finally the performance on both architectures with padding increases of about 60%.

Figure 5.17 shows the intra padding effects for data structure 7. As described above there is random inter padding used for the different bands. The assumption that there are only few conflicts inside the bands can be shown with this figure. One can also see that only if no padding is used the performance drops. For ascending padding values to the x– and y–direction the performance did not increase. For the Alpha 21264 CPU the performance also gets lower if the the padding values are set too large.

In comparison to data structure 3 in Figure 5.18 we can see data structure 8, where simulated inter padding is used. The performance rate has the same progression as in Figure 5.16. This is an indication for conflicts between the different bands.

Another method to prevent cache thrashing is also done by the hardware. For that the cache organization is implemented to the CPU using n–way set associative caches. E.g. the Pentium II CPU has a 4–way associative L2 cache. In Figure 5.15 we showed that for this architecture array padding almost does not lead to significant effects for the cache misses. This method is usually unmodifiable implemented in the CPU and thus cannot be influenced. The advantage for the Pentium II architecture in this example is, that the memory blocks can be mapped to four different cache lines each. This is the reason because also without padding the cache usage is much better as in comparison to the Alpha 21164 whose L3 off chip cache is direct mapped. Thus, it can occur that one data layout has different performance for several CPUs. For one CPU the thrashing effects can be prevented with the associativity of the cache unit, for other architectures which are e.g. direct mapped thrashing effects occur. For this array padding should always be implemented.

The method of array padding can also be used as a compiler technique [TCM94] [TR98] in order to prevent performance loss which may occur due to the described problem. But if the data layout is implemented in a disadvantage and cache inefficient way also array padding cannot significantly improve the performance. This effect can be seen for data structure 1 and 2. Inside this data layout the values are not arranged efficiently in terms of data locality aspects. With this also padding has no effect, because the data layout did not suffer from thrashing effects. It is still the task of the

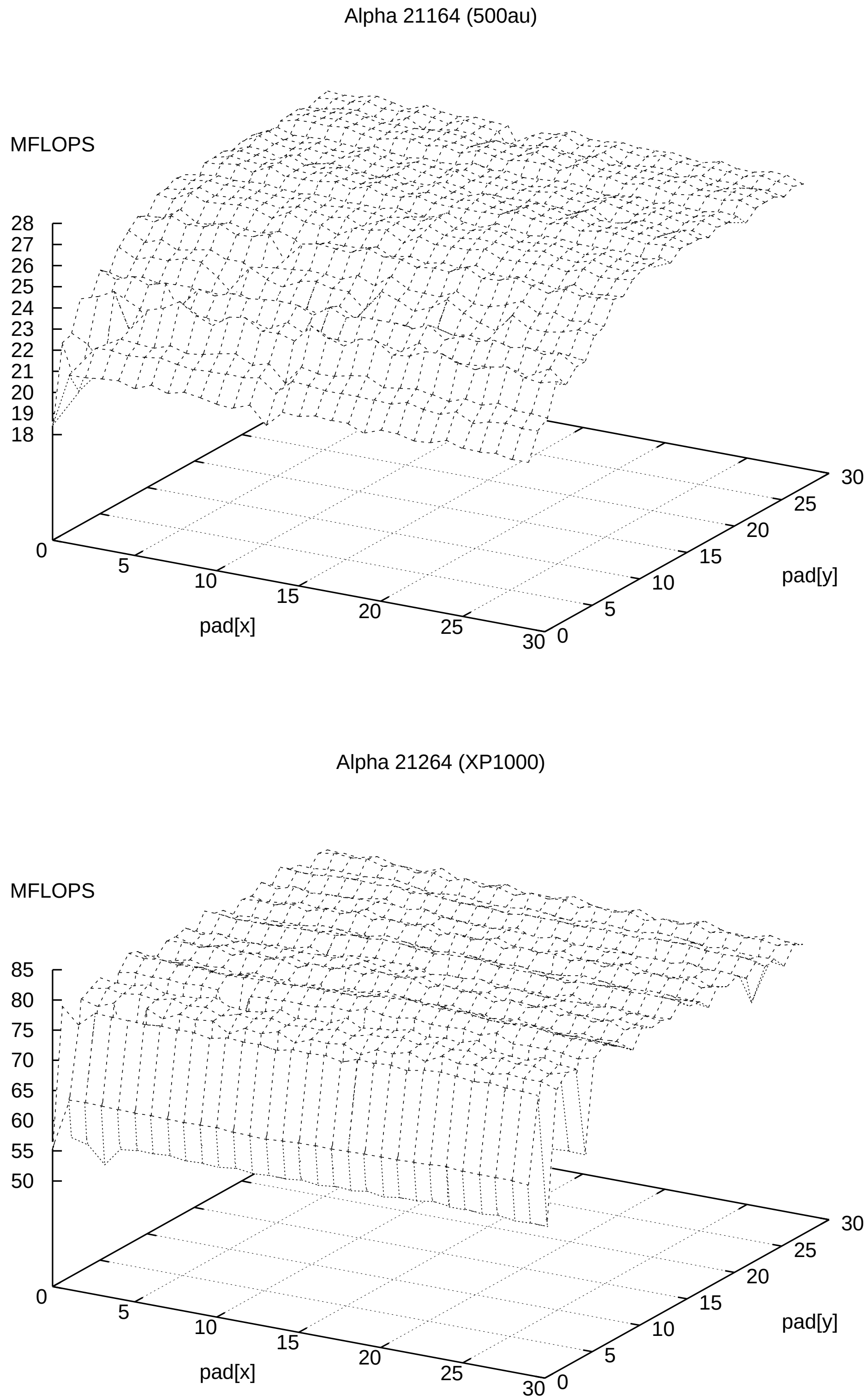
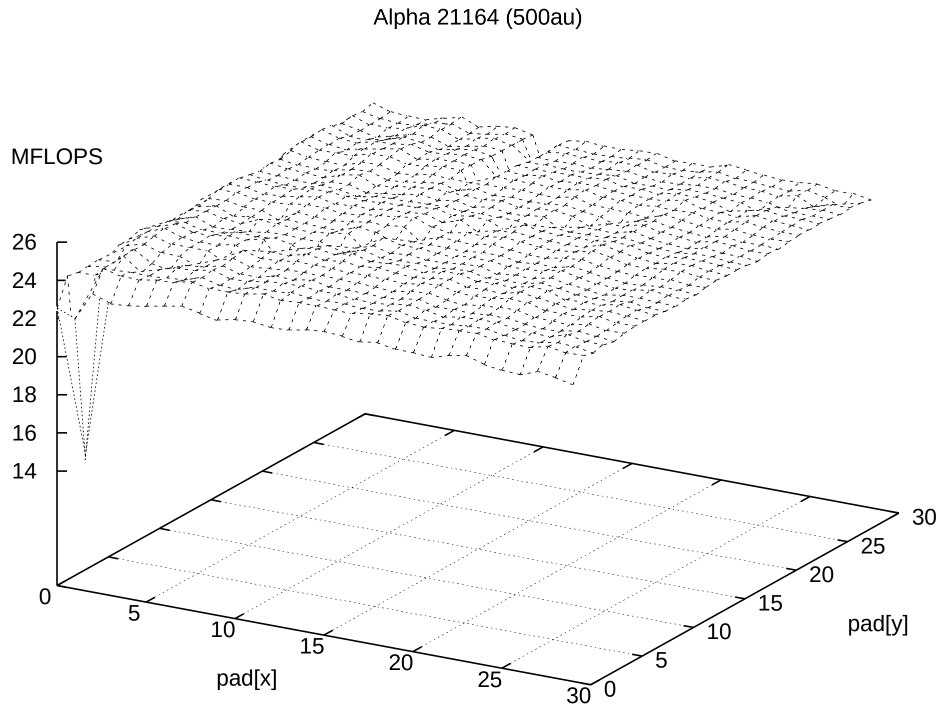


Figure 5.16: Padding example for data structure 3.



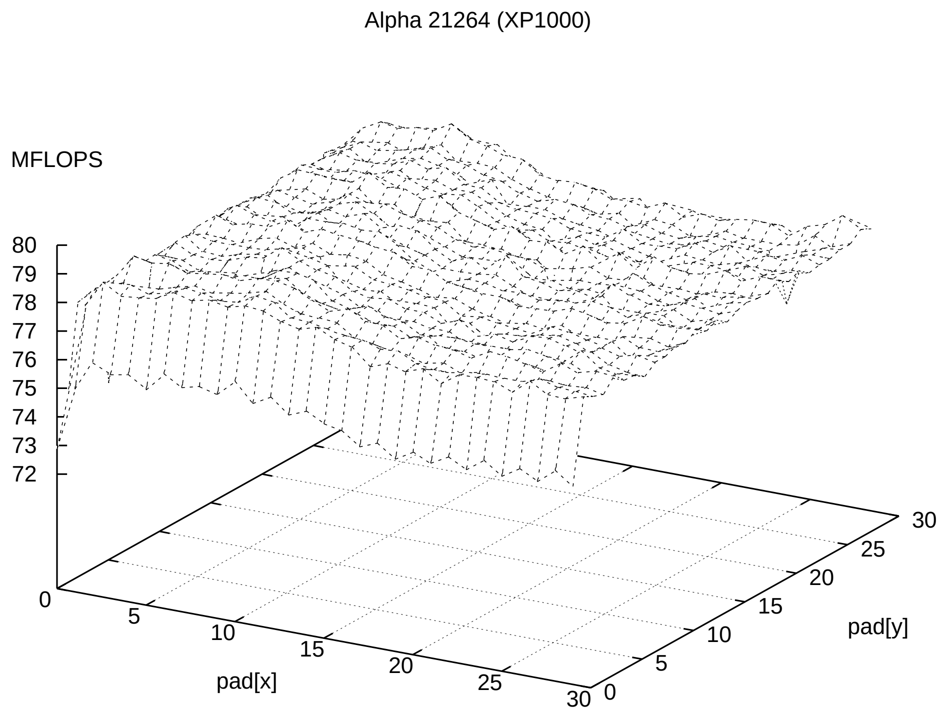
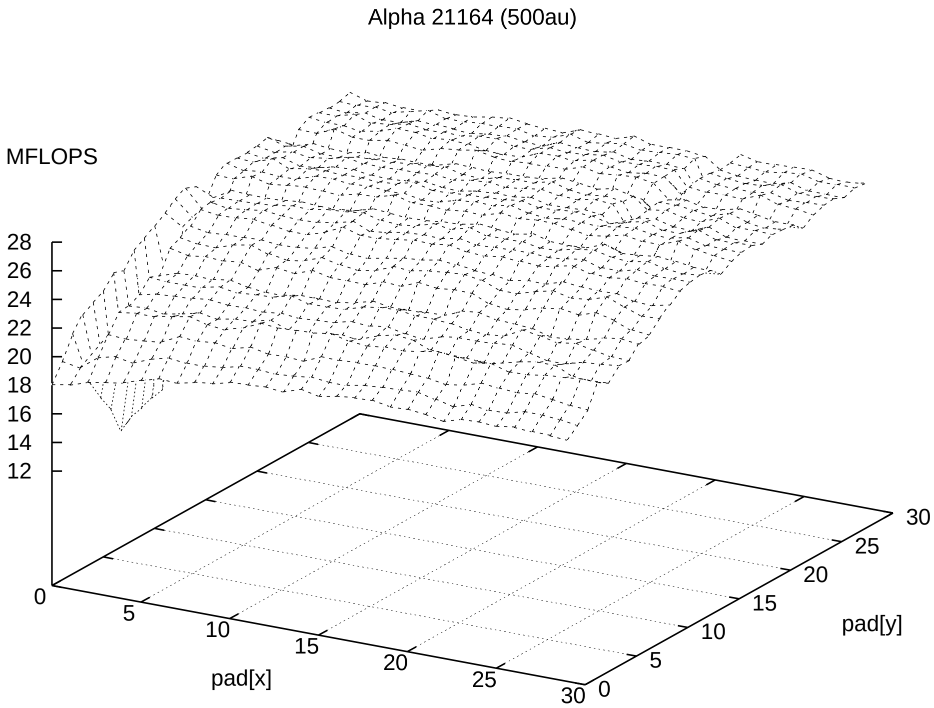


Figure 5.17: Padding example for data structure 7.



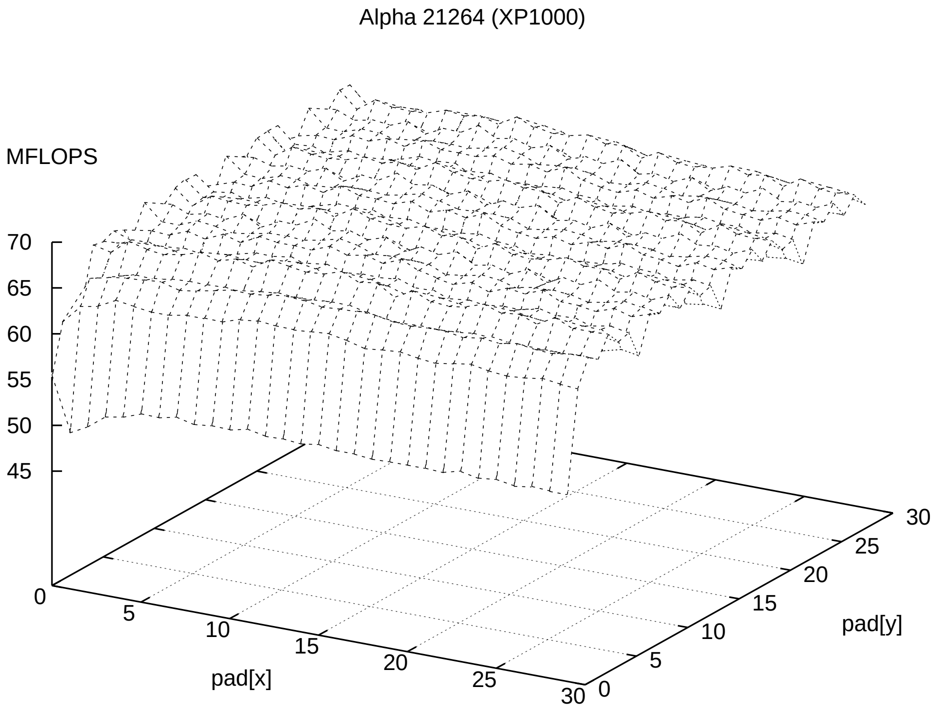


Figure 5.18: Padding example for data structure 8.

programmer to choose a data layout which has the data locality aspect in mind in order to gain good performance.

The examples above have shown that padding is very effective if a data layout has several conflicts. For this padding values used as intra or inter padding can help to prevent conflicts. In our examples only the intra padding is used. This reason for this is, that all data values are stored in one array. Also inter padding is useless for these examples, because of the used programming language C. In this programming language the distance between two arrays cannot be influenced by the programmer, because the array distance is influenced by various factors and therefore can be seen as kind of random.

Anymore padding values must be used with care. In the given examples we can see that not all padding values lead to performance increase. We have shown that when padding values are chosen too small or if they are used e.g. in the wrong position between the values, it may occur that they have no effect. Also if the padding values are chosen too large the performance drops, because with array padding also the resulting size of memory increases — which is shown in Figure 5.17 for the Alpha 21264 CPU.

5.2.3 Array Merging

The advantage of *array merging* is that with this method better data locality can be reached. For this reason, in our example for the relaxation method of Gauss Seidel, all needed data for one iteration should be stored locally in memory. With this improved data locality the resulting data layouts should be more resistant towards thrashing effects, even without array padding methods. Also the CPU specific cache organization should not lead to significant performance distinctions between several architectures.

To investigate these performance effects we implemented array merging for data structures 4, 5 and 6. All using different aspects in which way the values for the solution vector $\vec{u}_h$, the values for the coefficient matrix A and the right–hand side $\vec{f}_h$ should be merged. For that we measured the number of L2 cache misses that occur with and without the use of padding for these data layouts, which can be seen in Figure 5.19.

The results of the measurements are explained in the following:

- For data structure 4 the number of cache misses does not change with and without padding. Also fewer cache misses occur than e.g. for data structure 3. This leads to the assumption that the data is stored in a more efficient way in view of data locality aspects. The reason for this can be illustrated if we remind Figure 5.12 that shows the method of Gauss–Seidel. There all values of the solution vector $\vec{u}_h$ are referenced very intensively for one sweep through the grid in comparison to the right–hand side $\vec{f}_h$. If an access to a value of $\vec{f}_h$ occurs, also all according coefficients are needed. This is exactly what the Gauss–Seidel method — shown in Equation 2.9 — does for one iteration. Therefore $\vec{f}_h$ and the coefficients are merged together and $\vec{u}_h$ is stored separately in the beginning of the array. In comparison to this kind of storage, if the solution vector $\vec{u}_h$ is merged with the coefficients — as e.g. done in data structure 1 — the cache reuse decreases.
- For data structure 5 both values for the solution vector $\vec{u}_h$ and the right–hand side $\vec{f}_h$ are merged. The coefficients are also merged and stored behind. This leads to a higher cache miss rate than for data structure 4. But nevertheless the storage also is not so sensitive towards thrashing effects.
- In data structure 6 we have used the same storage scheme as in data structure 3 with the difference that the coefficients are merged and not stored in a band wise manner. This data layout is also affected by thrashing effects. The number of cache misses decreases if we use array padding. Possibly in this data layout $\vec{u}_h$ and $\vec{f}_h$ have conflicts. Due to this measurement results one might suppose that for data structure 3, also the different coefficients are in conflict with each other.

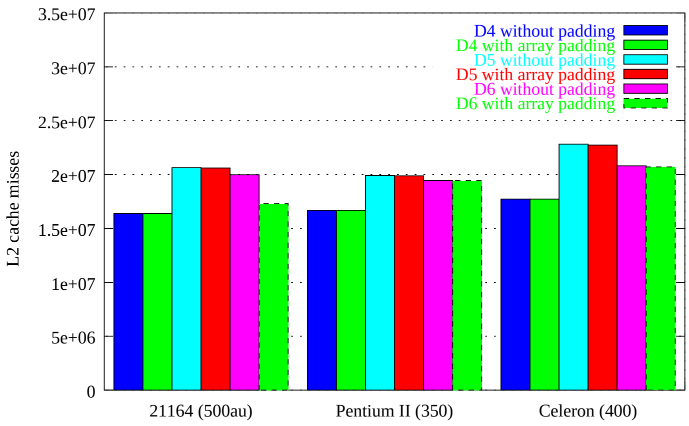


Figure 5.19: The number of L2 cache misses for array merging optimization.

For the data layout aspects it can be summarized that array padding and array merging can lead to high performance gain. If for large amounts of data the according data layout is chosen in an inefficient way in terms of locality aspects — so that thrashing effects can occur or the data locality is very sparse — this can lead to significant performance loss.

Figure 5.20 shows the resulting CPU time which is measured using 1024 grid intervals for each dimension, 10 V–cycles and two pre– and post–smoothing iterations with the red–black Gauss–Seidel method. Our 2–D multigrid program also used loop interchange optimization for the time measurements. We therefore concentrated on data structures 1, 3 and 4. For each data structure we measured the total time without padding compared with the best padding method that was found using a set of padding blocks for each architecture.

For data structure 1 array padding almost had no effects because the data layout suffers from a sparse data locality. This data layout can be seen as an example how data should not be arranged within our 2–D multigrid program. In data structure 3 the execution time is reduced with array padding.

The effect resulting from array merging can be seen for data layout 4. The data locality even is high without padding and thrashing effects do hardly occur, because also array padding did not lead to better performance.

For data structure 3 we have already shown that the best performance is achieved if array padding is done. This data structure has the best performance for the Alpha 21264 where its results are also significantly better than for data structure 4. Onto all other architectures its performance is almost equal or lower than that of data structure 4.

From these results we concentrated for further optimization on data structure 3 (bandwise storage) and data structure 4 (array merging), because these data structures provide the most efficient data storage that was found in our experiments.

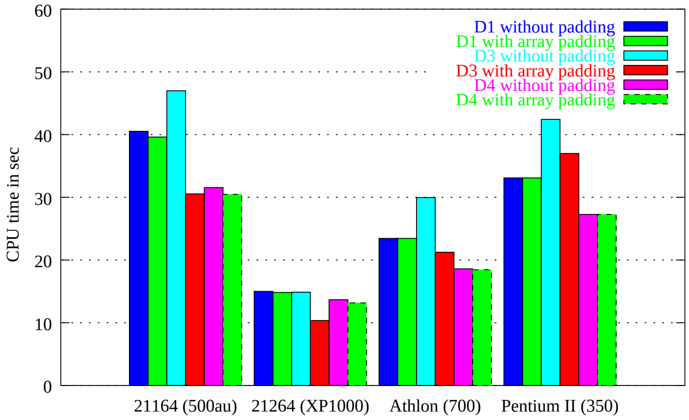


Figure 5.20: Overview of the CPU time for different data layouts.

5.3 Loop Fusion

The *loop fusion* optimization is used to combine several loops into one loop. This optimization method can be done with loops that e.g. have accesses to an array with the same index sets. The advantage of this method is that multiple loops suffer from more overhead than one single loop. This overhead can be assignments that have to be made within each loop and also tests if variables contain specified values. If there is only one loop used a lot of this overhead can be reduced. However the main advantage is that the memory usage is more efficient, because if the operations inside the loop access the same data — data which is already loaded into the cache — the performance increases. This results in faster accesses to the data leading to a much better cache reuse. Also the compiler can do loop unrolling much more efficiently [DS99].

The following shows an example, which is written in C code. There the both loops are fused, because they access the same index set:

```
for(i=1 ; i < 1000 ; i++){
   a[i] = b[i] + d[i];
}

for(i=1 ; i < 1000 ; i++){
   c[i] = b[i] + e[i];
}
```

Both loops can be fused:

```
for(i=1 ; i < 1000 ; i++){
   a[i] = b[i] + d[i];
   c[i] = b[i] + e[i];
}
```

In the previous example one can see that array `b[i]` can be reused in each step of the loop, which leads to a better cache reuse, mainly if the arrays include a large size of memory. Also in this example unrolling and prefetching can be used more efficiently.

In our multigrid program there are also two loops which are executed in sequential order. These two loops can be found in the `smooth()` function. The implemented red–black Gauss–Seidel smoother first relaxes all red points. After that another sweep through the grid points is done where all black points are relaxed. Figure 5.21 shows an example for a small grid that consists of 16 unknowns. In the figure we have shown two sweeps through the grid. The first sweep starts with the relaxation of the red points. If this is done the second sweep relaxes the black points.

In this example the data structure has a size of 1 KB and therefore would fits almost completely in the cache. But if we consider 2–D grids that consists of 1024 intervals for each dimension in one direction we have about 1 million unknowns. For this the data structure has a size of 56 MB. This amount of data is too big to fit in the cache levels of the architectures that were used within this thesis.

If we take a look at Figure 5.21 we can see that when in row number 2 the relaxation for the red points is done, the black points in row number 1 can be relaxed without violating any data dependencies occurring due to the red–black Gauss-Seidel smoother. For every row where the red points are relaxed the lower row with the black points can also be relaxed. If we remember the example in Figure 5.14, we can see that in one memory block which is brought to the cache not only the red points can be found. For data structure 4 e.g. this memory block includes all neighboring points. With this also some neighboring points of each red point are loaded. This data can be reused when the black points are relaxed in the line below.

This method should lead to a much better cache reuse as if the both red–black loops are executed separately, because for large grids this information would be replaced. We must only pay attention in the beginning and in the end of the loop. In the first row we can only relax the red points, because in the line below there are no black points which have to be relaxed because we assume Dirichlet boundary conditions. Also in the last row of of the grid we have to relax the black points separately, thus the data dependencies are not violated.

Our optimized relaxation loop starts in row two and ends in the last line. The only rows of the grid that are relaxed outside the loop are the first row (red points) in the beginning, and when the loop is finished we must additional relax the last row (black points).

The loop fusion optimization of our algorithm is measured using a 2–D grid with 1024 intervals for each dimension. In order to have a more detailed view of the performance increase we only relaxed the finest level with the Gauss–Seidel method. The smoothing is done twice, therefore two sweeps through the grid must be performed with the fused red–black iterations. Without loop fusion the number of sweeps through the grid doubles to four.

The results are shown in Figure 5.22. There a comparison between the version without loop fusion and with loop fusion can be found. As already explained in Section 5.2 we concentrated on data structure 3 and 4 for all further optimizations.

In the figure the MFLOPS rate for both data structures is shown. We used the best padding variables that were found for both data structures. For the Alpha 21164 we measured a performance increase for both data structures. The increase is by about 50%. The performance for the Alpha 21264 increases by about 30% for both data structures. For the Athlon and the Pentium II processor the performance improves by about 25%. Also one can see again that the best performance is reached using data structure 4 (array merging). Only for the Alpha 21264 data structure 3 (bandwise storage) has the best performance.

If we investigate the Pentium II measurements in more detail, one can see that the results for data structure 4 without loop fusion are also higher than the results for data structure 3 where the loop fusion technique is applied. This result verifies the advantage of the merged data layout which provides even more data locality for our code resulting in better performance. But the performance can also differ between the architectures for various data layouts. From this result we suppose that the Pentium II can handle data structure 4 in a more efficient way than data structure 3.

The performance between the slowest and the best result differs about 50%. In contrast to this for the Alpha 21264 data structure 3 provides the best results. For this reason we still measured both data structures — the bandwise storage and the merged data layout.

In Figure 5.23 we additionally showed the L2 cache misses that occur for the Alpha 21164 and the Pentium II architectures. For the Pentium II one can see that data structure 3 suffers from a lot of cache misses. When the loop fusion method is applied the cache misses are reduced by about 50%. The cache misses for data structure 4 are reduced by about 30%.

For the Alpha 21164 the L2 cache misses are reduced by about 10%. If we additionally compare the L1 cache misses for the Pentium II CPU with the L2 cache misses for the Alpha 21164 we can see

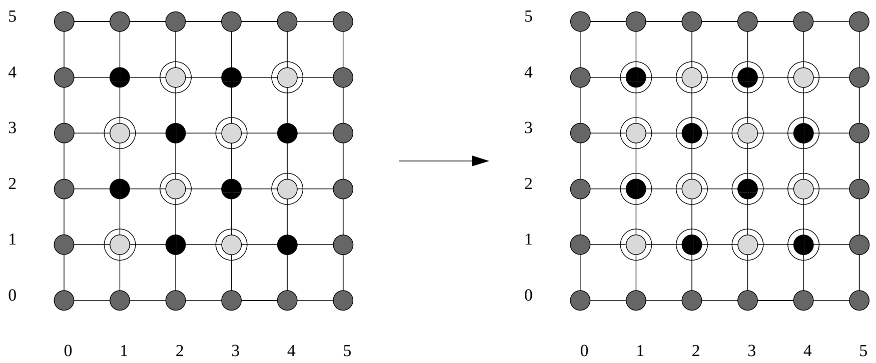


Figure 5.21: The red–black Gauss–Seidel method for a grid with 16 unknowns.

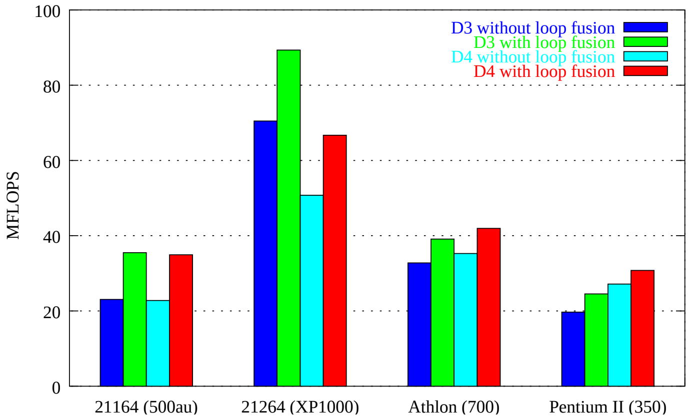


Figure 5.22: Comparison of the MFLOPS for the loop fusion method.

that the L1 cache misses are also reduced by roughly 10%. For that we suppose that the reduction for the L3 cache misses for the Alpha 21164 is similar to the L2 cache performance improvement for the Pentium II. Furthermore it should be noted that the decrease of the loop overhead also contributes to the the performance increase for the different architectures.

As seen in the previous results the improvement with the loop fusion optimization is between 25% and 50% for the architectures investigated in this thesis. This performance increase results if two loops are fused. For this we have shown that the cache miss rates and also the loop overhead could be reduced, because this technique leads to a more efficient cache usage and data locality.

In the next we show an enhanced version of this technique. The *1–D blocking* method that is further investigated can be compared with the fusion technique, because multiple lines are blocked.

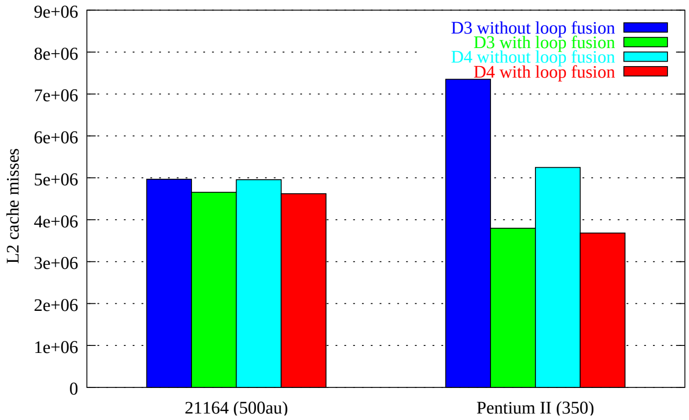


Figure 5.23: The L2 cache misses for the loop fusion optimization.

5.4 Blocking

The blocking optimization methods that are described in this section can be compared with the fusion method in the previous section. In comparison to the loop fusion method for the blocking optimization method multiple iterations are blocked with one sweep through the grid. For the blocking optimization the intention is to reuse data that is already in the cache and with it to minimize the number of cache misses citedimeSc99.

For our multigrid program the red–black Gauss–Seidel iteration is blocked, which means that multiple iterations can be done with one sweep through the grid.

In this section we investigate 1–D blocking and 2–D blocking. We start with the 1–D blocking method. There we blocked multiple grid lines. For the 2–D blocking method a 2–D block is moved through the grid. In the code of our program already all previous optimizations are integrated with the best performance that was found.

5.4.1 1–D Blocking

The first blocking method that is implemented in the multigrid program is the *1–D blocking*. With this method multiple red–black Gauss–Seidel iterations are blocked. In comparison for the loops fusion method where also two sweeps through the grid are blocked. We used one complete grid line for a 1–D block.

For the 5–point stencil that is used for our approximations, every grid point that is relaxed with the Gauss–Seidel method references his left and right and also its upper and lower neighboring grid point. With this also additional data which is located in the same memory block is loaded into a cache line. The blocking technique tries to reuse this additional data. Figure 5.24 shows in which way e.g. two red–black iterations can be blocked with this optimization.

The algorithm starts with line 1. In this line all red points are relaxed. Then the algorithm touches all red points in line 2. After that all black points in line 1 can be relaxed. In line 4 the figure shows how the algorithm works if the *setup phase* is done. The setup phase indicates that still not all not all operations for the 1–D blocking can be done, because if all lines below the first line would be accessed, these would be a reference outside the grid. The setup phase for this example ends in line 4. There all red points are relaxed, which is shown with the dotted circles. Then in the line below the black points are relaxed for the first time. In line 2 the red points can be relaxed for the second time, and in line 1 the black points are relaxed. When this is done this algorithm repeats for all lines above.

The algorithm thus needs four 1-D blocks — which exactly contains one grid line each — afterwards if two red–black iterations are blocked. We define this as a *working set*. The working set shows the number of points and therefore the amount of memory that must be held in the cache within the

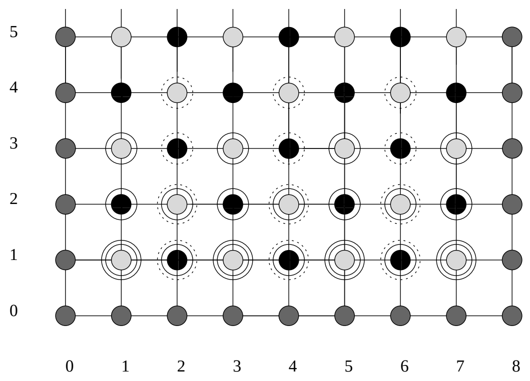


Figure 5.24: 1–D blocking schema with the red–black Gauss–Seidel method.

optimization method to avoid cache misses. The working set for 1–D blocking is defined as:

$$\text{Working set} = \text{number of iterations} * \text{number of grid points per line}$$

For the implementation one must remember that in the beginning the red points need a setup up phase and after the first 1–D block on the last grid line is relaxed also the lines below still have to be relaxed on this last grid line. This means that in the beginning the working set is enlarged, because for the first line no lines below exists. In the last line of the grid the working set is reduced, because all uppermost grid lines that have iterated the uppest grid line the relaxation is done for this 1–D block.

The easiest way is to move all 1–D blocks from the outside through the grid in the implementation, so only relaxation is done if the 1–D block is located inside the grid and contains unknowns. The algorithm for this can be seen in Figure 5.25 that shows the 1–D blocking optimization method on the basis of a Nassi–Schneiderman flowchart.

In order to show the efficiency of this algorithm we blocked two and four iterations of the red–black Gauss–Seidel smoother, for a grid size of 1024 intervals in each dimension. For this performance test only four red–black Gauss–Seidel iterations on the finest level are measured. Again the both most efficient data structures(3+4) that we have found are used. The results are shown in Figure 5.26. For the reason to avoid that thrashing effects influence our measurements additional suitable padding variables are inserted into both data structures.

The first observation one can see is when two iterations are blocked in comparison to Figure 5.22 the performance for the smoothing part increases by about 25% on all architectures. In the case when four iterations are blocked the performance gain is 15% for both Alpha architectures. For the Athlon architectures almost no increase can be found. The cause can be explained if we take a look at the working set that is used for a grid containing about one million unknowns. If two iterations are blocked, four complete 1–D blocks (four grid lines) are accessed within this working set.

Each grid line for this example can be divided into multiple memory blocks that are mapped to different cache lines. If all of these grid lines fit in the cache memory accesses are decreased. On the other hand if the working set is set too large so that additional cache misses occur, the reason for this can be found in the fact that the needed amount of data for the 1–D blocking algorithm does not fit in the cache any more.

If four iterations are blocked the working set increases to eight 1–D blocks (eight grid lines). Both Alpha 21264 architectures have 4 MB off chip cache. The Ahtlon in comparison to this has only 512 KB. If four iterations are blocked, the cache performance cannot be improved for the Athlon CPU. For that no speed increase is measured for this CPU.

The second observation is that also with this optimization the Alpha 21264 works better with data structure 3. Apparently this architecture can handle the data organization for data structure 3 in a more efficient way than for the merged data layout found in data structure 4, independent of the optimization method that is applied.

In Figure 5.27 the numbers of cache misses are shown. For the measurements we used the Alpha 21164, the Celeron and the Pentium II CPUs. In this figure we can see that for the Alpha 21164 the number of L2 cache misses could not be decreased. The cause for this maybe can be explained with the small L2 cache of this CPU. To have a better comparison we additionally measured the L1 cache misses for the Pentium II and the Celeron processor, which is almost equal in size:

	loop fusion	1–D blocking
Alpha 21164 (500au)	4.621.473	4.943.132
Pentium II (350)	7.353.359	7.366.150
Celeron (400)	7.360.510	7.361.182

In the above table one can see the L2 cache misses for the Alpha 21164 CPU and the L1 cache misses for the both Intel processors. The results are measured for a grid with about one million unknowns with the red–black Gauss Seidel method. We used two iteration s for both optimization

1–D blocking — With the 1–D blocking optimization technique multiple Gauss–Seidel steps can be blocked.

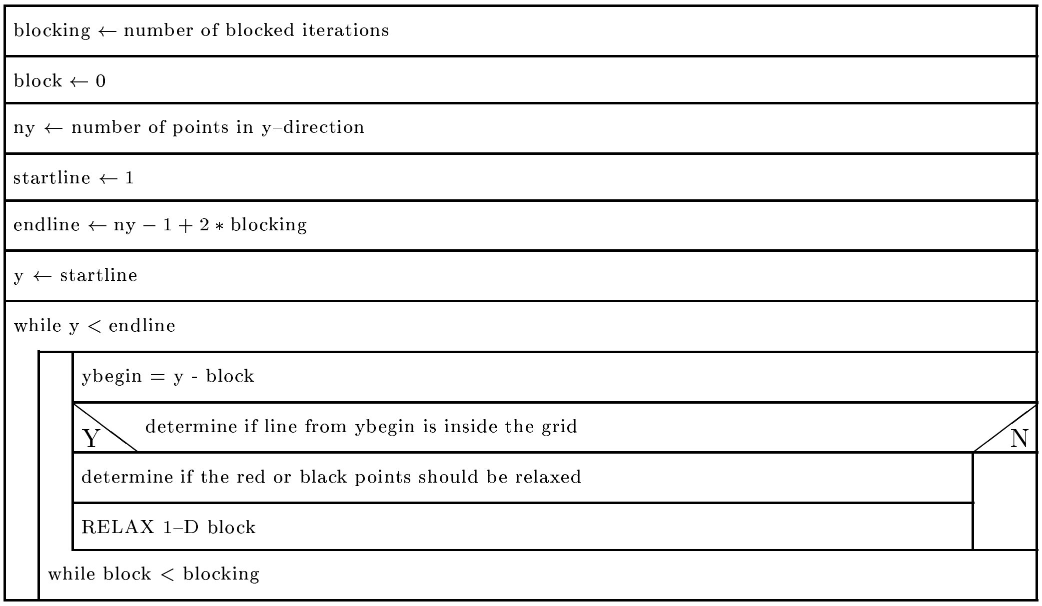


Figure 5.25: 1–D blocking algorithm.

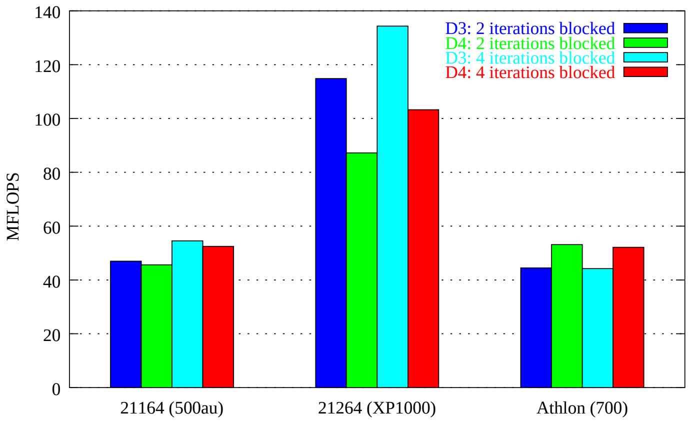


Figure 5.26: MFLOPS rates for the 1–D blocking optimization.

variants with two Gauss–Seidel steps blocked. For the 1–D blocking method the number of cache misses could not be reduced on all architectures. The cause for this maybe can be found in the small caches in comparison to the larger working set. The working set for the integrated 1–D blocking method was too large to fit in the caches of the CPUs under considerations. Another fact that we can see is that the L1 cache misses between the Pentium II and the Celeron did not differ. This is because both CPUs have the same processor architectures. Only the L2 cache size differs. Therefore the L1 cache misses are equal for both CPUs, which is verified with these measurements.

In Figure 5.27 we can also see, that for the Celeron CPU the working set also did not fit in the L2 cache. For this reason the performance is in difference to the Pentium II lower. The larger L3 cache that the Alpha 21164 has leads also to much fewer cache misses than for the Pentium II and for this the performance for the Alpha 21164 is better.

In this section we have seen that the implemented 1–D blocking method can lead to performance increase. But if for larger grids multiple iterations are blocked, our 1–D blocking optimization suffers from the growing size of the working set that arises when multiple lines or multiple iterations are blocked, because our 1–D block always contains one complete grid line. For this the performance cannot be improved further and the numbers of cache misses increase, because the cache sizes of the most architectures are too small to hold multiple 1–D blocks for grid larger sizes.

For further optimizations we have to decrease the working set in comparison to the 1–D blocking method so that the cache misses for the larger caches can be reduced farther. For this we implemented the 2–D blocking method which is investigated in the next section. For the 2–D blocking method the 2–D block is more variable in size leading to a smaller working set with which the cache can be used more efficiently.

5.4.2 2–D Blocking

In this section the 2–D blocking method is described. For this blocking algorithm multiple 2–D blocks are moved through the grid. Depending on the number of iterations that are blocked for every red–black Gauss–Seidel iteration two 2–D blocks are needed. They are placed in the grid so that in the upper 2–D block all red points are relaxed first. After that is done in the second blocked — that is placed one point below and one point to the left in the grid — all black points are relaxed.

In comparison to the 1–D blocking method the blocks can have a smaller working set than for the 1–D block which contains one complete grid line, because the 2–D block size can be set variable in each dimension. This should lead to a better performance. The goal for the performance measurements is also to find an optimized block size to the x– and y–direction. For the following experiments we again concentrate on data structure 3 and 4, using a constant padding size to prevent thrashing effects.

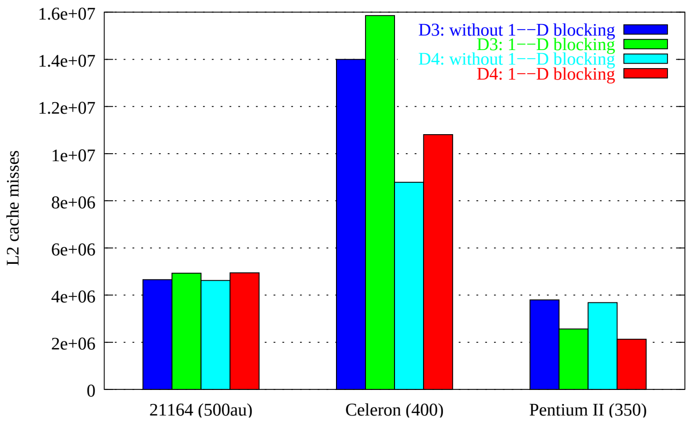


Figure 5.27: L2 cache misses for the 1–D blocking optimization.

In Figure 5.28 we illustrated the basic principle how the 2–D block is moved through the grid. In this figure the block has a size of six points to the x–direction and four points to the y–direction. The relaxation starts in the lower left corner, beginning with the first unknown point in the grid which is labeled $u(1,1)$. Starting form this grid point we placed the 2–D block over the grid points. Inside this block all red points are smoothed. The already smoothed points are marked with a circle. In order to smooth the black points another block – which starts one point to the left an one point below from the first block — is placed on the grid. In order to keep the data dependencies for the red–black Gauss–Seidel smoother the black points can foremost be relaxed if the red points in the upper 2–D block are relaxed. In the figure the right grid shows this procedure. If the 2–D block reaches the grid boundary is must be resized.

If all red and black points inside the two grids are smoothed both blocks are moved to the right starting exactly with the next unsmoothed point to the x–direction. There the red points are iterated first followed by the black points.

The 2–D blocking version that is implemented in our program used a 2–D block with block sizes that can be set variable to the x– and y–direction in the beginning of the optimization. For one red–black Gauss–Seidel iteration two 2–D blocks are needed which are moved through the grid. One 2–D block which relaxes the red and one 2–D block that relaxes the black points.

We have implemented the 2–D blocking optimization with different 2–D blocking versions. The difference for all implementations is generally the boundary treatment at the 2–D block which is implemented using different approaches. The first and second optimization methods only blocked 1 iteration. The third version also can block multiple iterations.

For the first 2–D blocking version the 2–D block size is determined in every step, in order to show if a boundary from the grid is reached. With this version one iteration is blocked with one sweep through the grid. We measured the performance for different block sizes to the x– and y–direction.

For the second version of the 2–D algorithm additionally a setup phase is done so that for the beginning of the 2–D blocking both 2–D blocks completely fit into the grid. With this version also only one iteration can be blocked. In advance to the first version also a lot of `if`–conditions are eliminated which are needed for the resizing of the 2–D block if a boundary of the grid is reached. In advance to the first also lower row and the left column of the red points are relaxed first in the setup phase. With that the blocking can start at grid point$u(2,2)$. The first version starts at $u(1,1)$ and therefore the second block which covers the black points must be resized to fit in the grid at the beginning. With the second version of our implementation both blocks completely fit in the grid at the beginning, and therefore only the 2–D block has to be resized only for the upper and the right boundaries of the grid. This version eliminates a lot of `if`–conditions that are needed in version 1 of the 2–D blocking algorithm needed for the 2–D block resizing.

In Figure 5.29 our both 2–D blocking versions 1 and 2 are measured. In order to show the performance that the `smooth()` function receives from this optimization, we again measured four Gauss–Seidel iterations on a grid with about one million unknowns. We used the both Alpha CPUs and the AMD Athlon processor to measure the performance for various sizes of the 2–D block. The

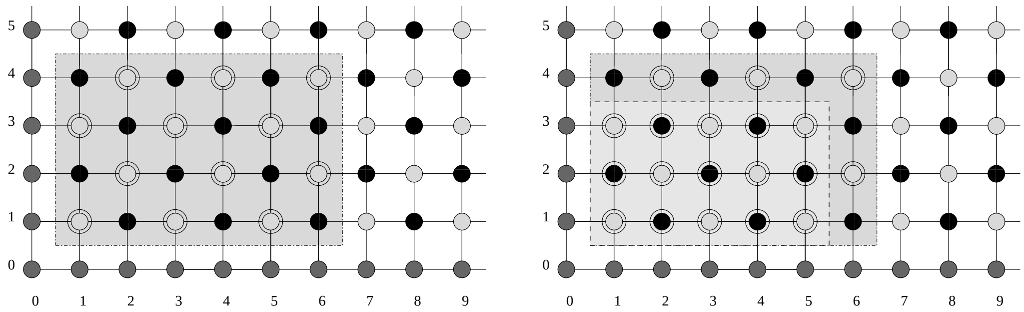


Figure 5.28: 2–D blocking: The grid on the left shows the 2–D block denoting the red points to be smoothed. In the grid on the right the black points are relaxed in the 2–D block below.

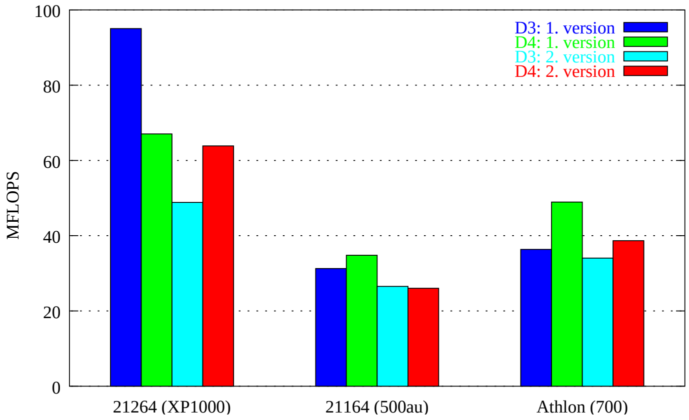


Figure 5.29: MFLOPS rates for the 2–D blocking optimization with the first and second 2–D blocking version.

best results for different block sizes to the x– and y–directions are taken.

In the figure we can see that for the Alpha 21264 again data structure 3 provides the best results in comparison to the both other architectures, where data structure 4 has the best performance. In comparison to Figure 5.26 we can see that the results for our first and the second blocking version have no performance increase in comparison to the 1–D blocking implementation. The results is even lower than for the 1–D blocking method. The cause for this result maybe the fact that only one iteration is blocked. Therefore the data reuse is also lower than if multiple iterations could be blocked. Also the computation effort for this implementation is more expensive than for the 1–D blocking method. In each step the 2–D block must determine if the 2–D block fits in the grid for its x– and y–direction or has to be resized. In comparison to the 1–D blocking eventually also fewer unrolling and prefetching could be done by the compiler, because of the structure of the code.

The second version where a lot of `if`–conditions are eliminated also did not receive a better performance than the first version of the 2–D blocking method. The performance even is lower on all architectures. Maybe also with this code the compiler cannot efficient apply optimization methods which are e.g. loop unrolling or prefetching. Generally the performance rate for both 2–D versions verifies the assumption that the compiler also could not take advantage of the second version of the 2–D blocking algorithm and therefore optimize the machine code.

For this we implemented a third 2–D blocking version with the following characteristics:

- The algorithm must be able to block multiple iteration, which is leading to a better data reuse, because for that data accesses are more local. This should decrease the number of cache misses.
- The source code of the algorithm must be arranged more efficient for the compiler so that it can perform loop unrolling more effective than for the first and the second version of the 2–D blocking algorithm.

With these characteristics the third implementation of the 2–D blocking optimization is done. The idea of the implementation is similar to the presented algorithm in Figure 5.25. The algorithm additionally must also take care about the x– and y–direction of the block size. Also the 2–D block is moved horizontally through the grid and then moved to the next y–position. For each iteration also two 2–D blocks are needed that are placed on the grid as showed in Figure 5.28.

Every block is shifted one point to the left and one point down in relation to the next upper block. This is done to keep the data dependencies for the red–black Gauss–Seidel iteration. Also the algorithm has to resize the 2–D block if boundary parts are reached and therefore the 2–D block does not completely fit into the grid.

If multiple iterations are blocked the lowest block e.g. starts outside the grid — the block is virtually placed outside. Therefore in this block no relaxation is done until the 2–D block is moved inside the grid. When the most upper block leaves the grid in y–direction the lowest block has to relax its according unknowns if it is still inside the grid. This is done in order to keep the data dependencies between the blocked iterations.

In Figure 5.30 the basic principle of our third version of the 2–D blocking optimization algorithm is shown.

In addition to this optimization method we also implemented a fourth version of this optimization method. We therefore extended the third version with the intention to help the compiler performing optimization to the code. Therefore the fourth version is split into three parts: The lower part, the interior middle part and the upper part.

The interior middle part can additionally be divided in the left and right boundary parts. These are the parts where the program has to resize the 2–D blocks because in this parts they do not fit completely in the grid. In the middle part all blocks that are used to relax the unknowns completely fit into the grid. The size of this middle part differs with the size of the used blocks, and also depends on the number of blocked iterations. If the block is placed in the middle interior part no boundary conditions have to be checked for the 2–D block. With this unrolling optimization can be made more effectively by the the compiler.

The performance results of the third and the fourth version of the 2–D blocking can be found in Figure 5.31. Again we used a grid size about one million unknowns. Only the smoothing time of four red–black Gauss–Seidel iteration on the finest grid is measured, with two iterations blocked. Also different block sizes are used whereof the best results are taken.

In comparison to Figure 5.29 one can see in the figure that the third and the fourth optimization method almost have the same MFLOPS rate. The performance in contrast to the first and the second version of the 2–D blocking optimization where only one iteration is blocked increased.

But the performance still is not better than for 1–D blocking. Therefore we additionally blocked four iterations in order to investigate the impact on the performance.

In Figure 5.32 and Figure 5.33 the same algorithm is measured with the difference that four iterations are blocked. Again only the finest level is iterated with red–black Gauss–Seidel method using four iterations which are blocked by the both optimization methods.

In this figure one can see that for the Alpha 21264 the MFLOPS rate increases from 102 MFLOPS up to 130 MFLOPS. This is an increase of about 30% in comparison if only two iterations are blocked. For the Alpha 21164 the increase is even of about 50% which is also the highest increase. For the other architectures the MFLOPS rate increases by about 30%.

A final comparison between the 2–D blocking and the 1–D blocking leaves different receptions. For the both Alpha architectures the MFLOPS rate only increases by about 10%. For the other architectures the increase is of about 40%. This fact leads to the presumption that maybe the compiler for the both Alpha architectures can handle the 1–D blocking algorithm even in a comparable way in sight of loop unrolling as the compiler on the other architectures. This fact also does not surprise because the compilers for the Alpha architectures are optimized for this CPU. In contrast to the architectures where the gcc compiler is used which is not optimized in such a special way for these different CPUs.

On the other hand the large off chip caches that the Alpha CPUs have maybe also influences this result. The off chip cache for the both Alpha architectures was 4 MB of size. The Pentium II and the Athlon in comparison only had 512 KB. Therefore the caches for the both Alpha CPUs can handle about eight times more data than the caches for the other CPUs. For this the both Alpha architectures suffer from lower cache misses than the other CPUs used within this thesis.

2–D blocking — In the 2–D blocking algorithm multiple red–black Gauss–Seidel iterations can be blocked

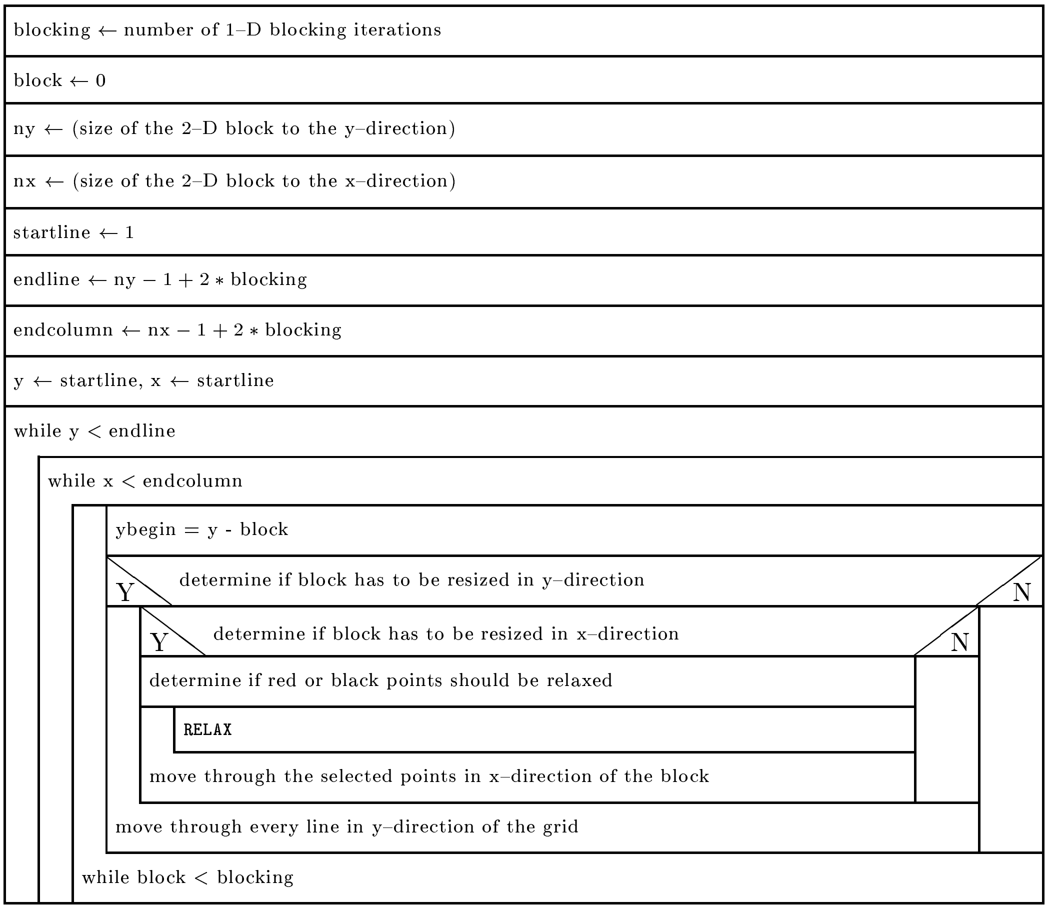


Figure 5.30: The third version of our 2–D blocking algorithm.

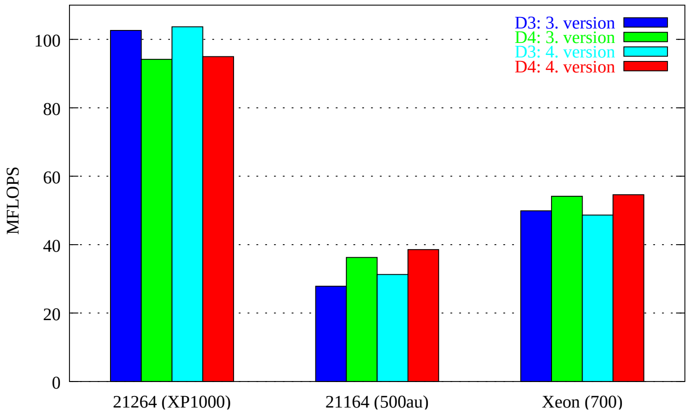


Figure 5.31: MFLOPS rates for the third and the fourth implementation of the 2–D blocking optimization with two iterations blocked.

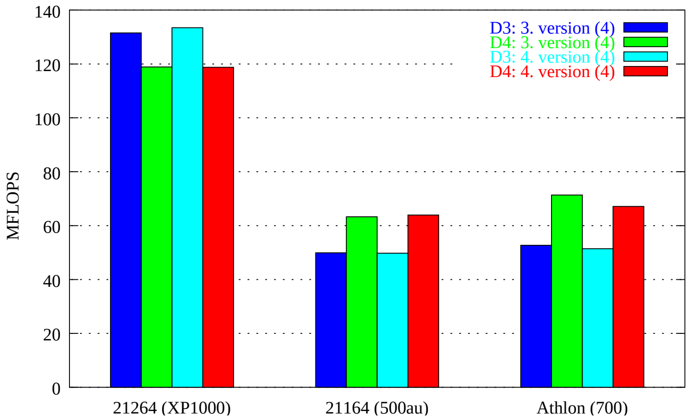


Figure 5.32: MFLOPS rates for the third and the fourth implementation of the 2–D blocking optimization with four iterations blocked for the first set of architectures.

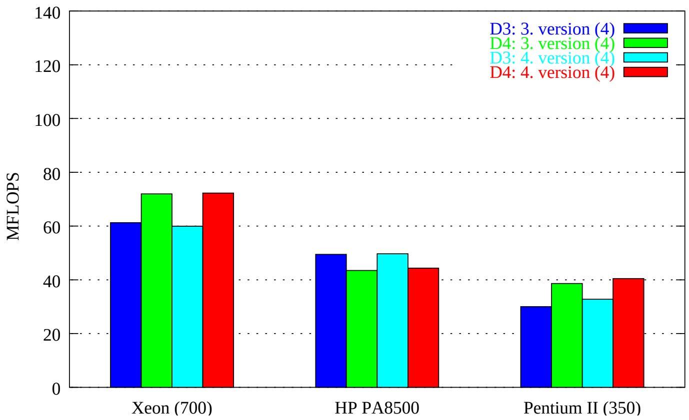


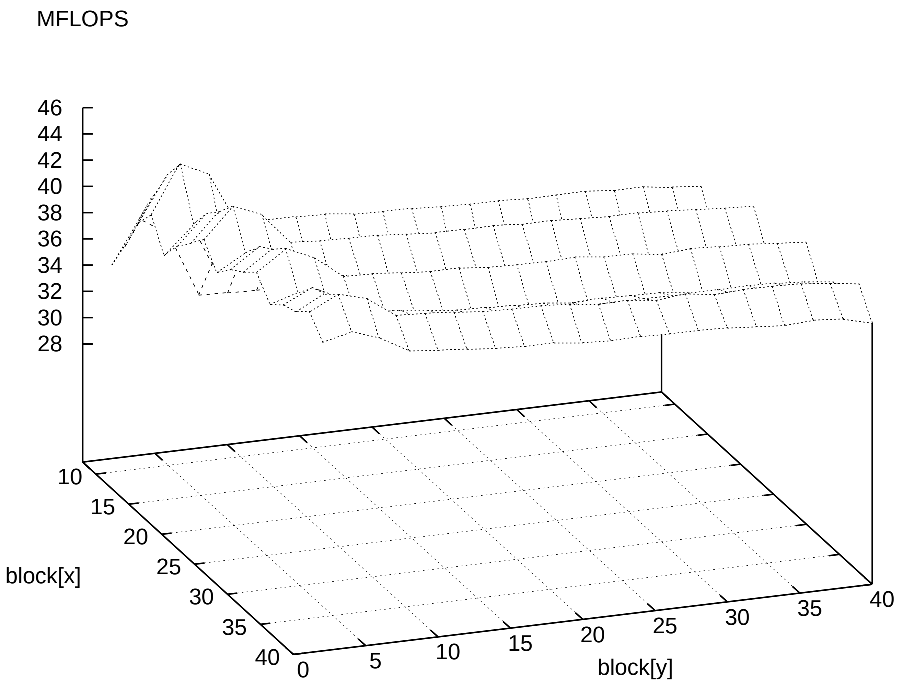
Figure 5.33: MFLOPS rates for the third and the fourth implementation of the 2–D blocking optimization with four iterations blocked for the second set of architectures.

At the end of our 2–D performance measurements we show an overview of the influences the different 2–D block sizes to the x– and y–directions have on the MFLOPS rates. In Figure 5.34 we have measured the MFLOPS rate for the Alpha 21164 on a grid with about one million unknowns. The 2–D blocking is used with four iterations blocked. The finest grid was treated using the red–black Gauss–Seidel smoother. Also we only smoothed the finest level.

In the figure the upper picture shows the MFLOPS rates for data structure 3. In the lower picture data structure 4 is shown. We started with a block size of eight points to the x–direction, because fewer points in x–direction do not make sense if multiple iterations are blocked. Also if the block is chosen too small the cache efficiency reduces and also the efficiency of unrolling optimizations. The performance also only raised if the block is made larger to the x–direction. But if the 2–D block is set too large the performance gain decreases.

The figure demonstrates that for the 2–D blocking method the MFLOPS rate is very sensitive towards different x– and y–values. For data structure 3 the MFLOPS rate is highest when the 2–D block to the x–direction is chosen about 25 points. The block size to the y–direction does not influence the performance significantly.

In comparison to this for data structure 4 if the block in y–direction is chosen to large the performance drops significantly. This leads to the presumption that the different lines suffer from thrashing effects or mainly the working set gets too large to completely fit into the grid. We have received the best performance for data structure 4 if the 2–D block is set to about 20 points in x–direction. The height of the block should be much smaller and was about two or four points to the y–direction to receive the best performance. This proposition was also validated for different architectures that are used within this thesis. For the Alpha 21264 data structure 3 provided the best results. With almost the same block size which is chosen for the other architectures.



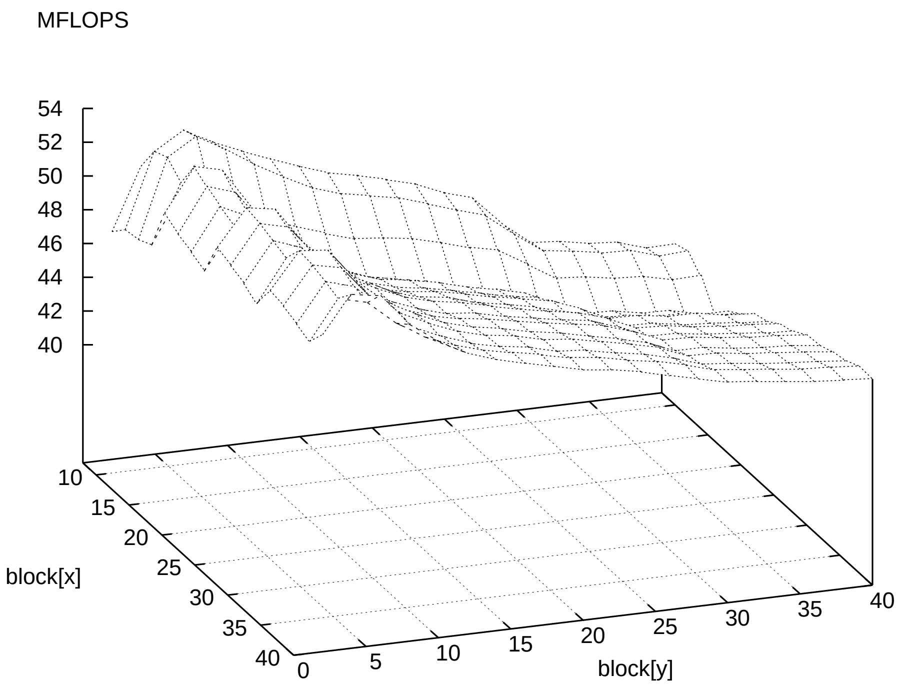


Figure 5.34: 2–D performance results for different block sizes on the Alpha 21164. In the upper picture the results for data structure 3 and in the lower picture the results for data structure 4 are shown.

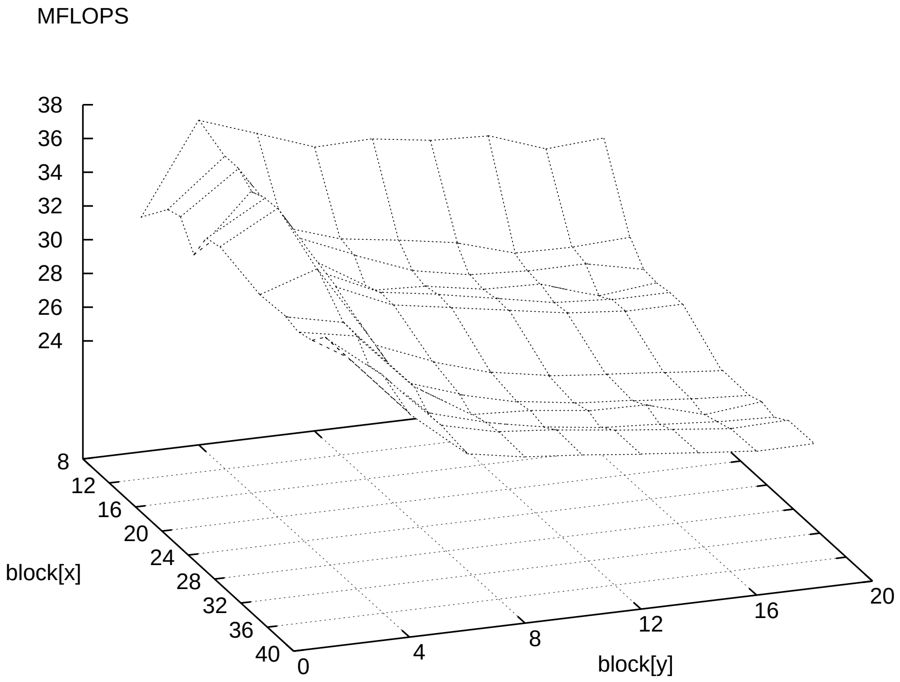
Additionally we also made a 2–D blocking example for different 2–D block sizes for the Hitachi SR 8000. The results can be seen in Figure 5.35. We measured the MFLOPS rate for a grid size with a grid 1024 intervals to each direction. With it we have about a grid with about one million unknowns. This leads to a memory size of about 56 MB. The architecture of the Hitachi SR8000 is explained in Chapter 3. For the measurement we reserved one node exclusively and computed the results of our multigrid program with one processor. The computation thus worked only sequential because no parallelization is used for our implemented code. We measured the performance of four red–black Gauss–Seidel iterations on the finest level with four iterations blocked.

In the shown figure data structure 3 is shown above, data structure 4 is found below. The best results that are measured on the Hitachi SR8000 used data structure 4.

In the above picture one can see that the MFLOPS rate decreased with the block size to the x–direction. If the block to the y–direction is set too large the MFLOPS rate also braked. Only for small blocks in x–direction the MFLOPS rate stayed constant to the y–direction. In comparison to the lower picture that shows data structure 4, data structure 3 in the upper picture has the lower performance. For large block sizes to each direction the performance breaks. This effect results because the 2–D working set is set too large to fit in the cache. The cache cannot be used efficiently for these 2–D block sizes. The picture for data structure 4 also verifies this. Nevertheless the performance is even low for the best 2–D block variables found. Maybe this occurs from thrashing effects that occur or the data layout is not so efficient in terms of data locality. Due to instabilities for the C compiler on the Hitachi SR8000 we could not perform further measurements.

In data structure 4 one can see that the MFLOPS rate increases with the block size to the y–direction. This is a contrarily effect as for data structure 3. Only if the 2–D block size in x–direction is set too large the MFLOPS rate breaks. This effect can be explained with the growing working set. If the amount of data cannot be cached anymore the performance breaks. This is the same effect as for data structure 3.

This performance example can also be seen as an impulse if the implemented multigrid program is to be parallelized. For parallel programming it is generally useful to keep the communication effort low. If additionally other CPUs have to wait because one processor e.g. suffers from cache misses because the 2–D blocking method is implemented in an inefficient way, the performance of the whole parallel program breaks. Therefore also for parallel computing — for which the Hitachi SR8000 is generally developed — an efficient cache organization for one CPU is also very important.



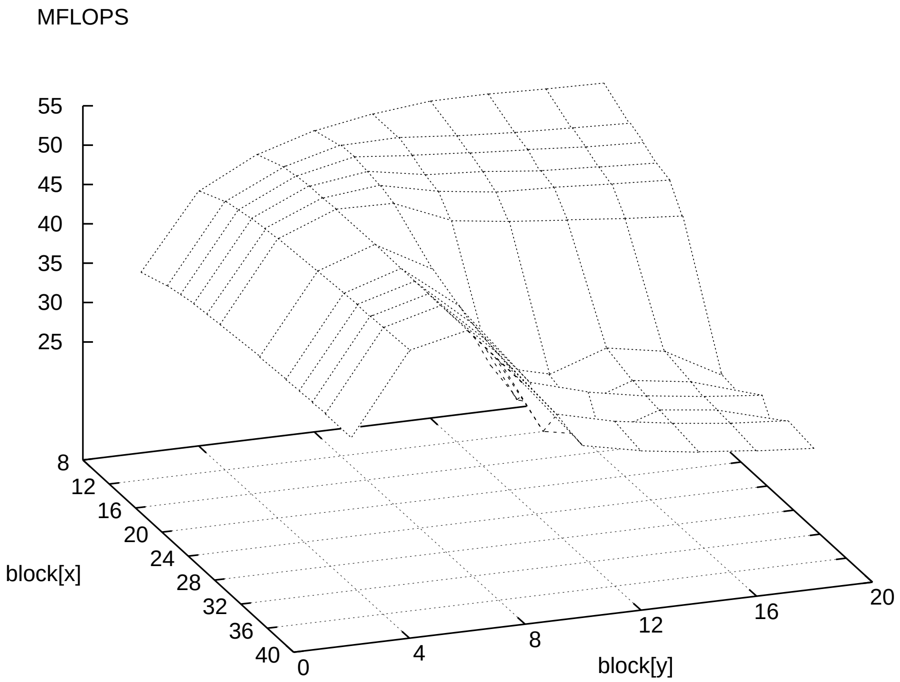


Figure 5.35: The MFLOPS rates for the Hitachi SR8000. The 2–D blocking method is used with different block sizes in x– and y–direction. The upper figure corresponds to data structure 3 and the lower figure refers to data structure 4.

5.5 Comparison of MFLOPS between different grid sizes

In this section we show the influence of the cache size for the performance of our multigrid program using different datasets. The memory requirements for the different grid points can be found in Figure 5.36. There the number of intervals to each direction is listed with the according size of data in MB. Therefore we measured the MFLOPS rate on one special architecture, the Intel Xeon processor with 1 MB of L2 cache.

The 1–D blocking optimization is measured in comparison to the 2–D blocking method. Due to the large main memory of about 2 GB which the Xeon–based machine has, also grids with 2048 and 4096 intervals in each dimension could be computed. These are grids containing about 4 million and 16 million unknowns. Again only the `smooth()` function was examined on the finest grid to show the influences that come with the different optimization methods. The red–black Gauss–Seidel smoother is iterated four times. The 1–D blocking algorithm blocks two and the 2–D blocking method blocks four iterations with one sweep through the grid.

In Figure 5.37 the MFLOPS rate for different grids are shown. For every grid point we measured the performance for the 1–D and the 2–D blocking method. Starting with 16 intervals (which means 225 unknowns) we doubled the number of intervals in each direction until 4096 intervals in each direction are used.

The first observation is that the MFLOPS rate decreases as the number of grid intervals increases. The cause is that the amount of data increases and does not fit in the L1 cache any more. This effect can be seen when the interval size increases from 64 to 128. After that with the number of intervals increasing to 256, also the L2 cache could not hold the complete amount of data any more. In Table 5.36 we can see that increasing the grid size form 64 to 128 intervals the resulting data size with about 0.9 MB is too large to fit in the L1 cache of the Xeon. Again, with 256 intervals the data set of 3.9 MB does not fit in the L2 cache any more.

If both optimization methods are compared the 1–D blocking method has a better performance for grids that are smaller than 128 intervals in each direction. After that for intervals equal and larger than 128 intervals the 2–D blocking optimization method has a better performance than the 1–D blocking method. The cause for this can be found in the smaller working set that the 2–D blocking method has in comparison to the 1–D blocking. As we have seen in the previous section for 1–D blocking if the grid line size gets too big to completely fit into the cache the performance stalls. Therefore the 2–D blocking methods have their advantage because the block size can be chosen in a way that the cache lines are used more effectively. For the 2–D blocking optimization this results in a better cache reuse and even fewer cache misses. At this point the additional computation effort that has to be made for the 2–D blocking method has its advantage in comparison to the 1–D blocking method.

For grids containing only a few unknowns almost no optimization is needed if the data sets completely fit in the caches. In the figure one can see that for very small grids about 225 unknowns (16 intervals) both optimization methods fail and the MFLOPS rate provides the lowest result over

intervals per dimension	size in MB
16	0.02
32	0.06
64	0.23
128	0.9
256	3.5
512	14
1024	56
2048	224
4096	896

Figure 5.36: The amount of data for different grid sizes.

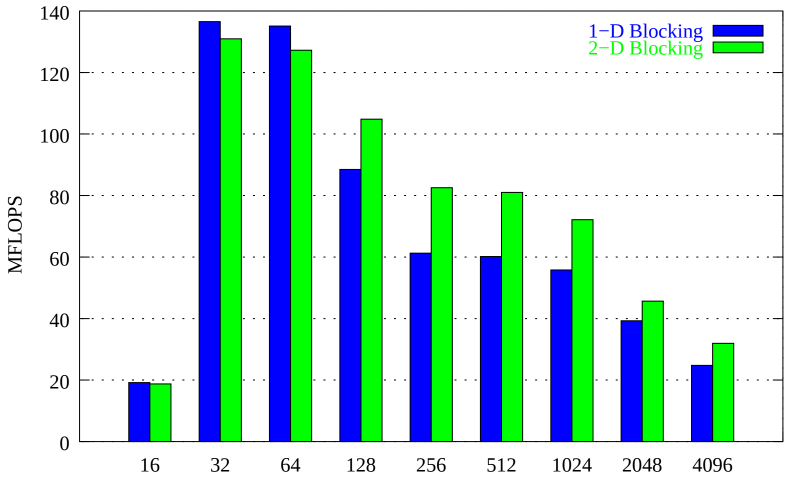


Figure 5.37: MFLOPS rates for the Intel Xeon processor for different grid sizes.

all measured grid sizes. Therefore the performance suffers from the additional computation effort that was made in order to increase the working set. Also the advantage of loop unrolling effects is reduced because the grid lines are very small in size.

For large grids that do not fit in the cache levels the blocking methods with the smaller working sets have advantages. More data can be referenced locally in memory which leads to fewer cache misses than without optimization. Also when the number of grid points increases the compiler can use loop unrolling in a more effective way.

5.6 Case Study: Overall performance of the Alpha 21164

In this section we performed a case study of the Alpha 21164 using the *Compaq* (formerly Digital) *Continuous Profiling Infrastructure (DCPI)* tools that is explained in Section 4.4.2. This section has the intention to exemplary show in which way a detailed program and source code profiling can be done. With an example we show how to find the bottlenecks which are responsible for the performance loss inside a program.

The profiling tools also show the machine code which is produced by the compiler. With it the instruction which produces the top stalls can be found. The DPCI profiling tool includes a lot of programs with which such an analysis could be performed. It should be clear that this profiling is very machine specific and only concentrates on one architecture. But for an exemplary profiling demonstration this restriction can be made.

In the following analysis we held the best optimization that was found with our implemented multigrid program, which is the 2–D blocking method. Also all other optimization values were hold inside the algorithm. This is constant padding size to prevent thrashing effects that possibly could influence the performance results. For the data layout we used data structure 4. For the blocking size we used a block size with 40 points in x–direction and 4 points to the y–direction. We blocked four iterations with one sweep through the grid. As optimization method we used the third version of our 2–D optimization method which provides the best measured performance of all our optimizations.

With it the profiling data gets accurate enough, we performed 200 Gauss–Seidel iterations on the finest level with a grid size of about one million unknowns – this correlates with a line size about 1025 points in each dimension of the 2–D grid.

We started analyzing the source code using the `dcpilist` tool. For the first measurements the source code was compiled without any compiler optimizations. We measured the number of board–level cache misses and the number of cycles that are spent in each iteration.

With the `dcpilist` tool the output shows the according events for each line. In our example each line starts with the number of L3 cache misses (board–level cache misses) in the first column and the number of cycles spent for each instruction in the second column. The output can be found in Figure 5.38.

Figure 5.38 shows the source code for the third version of the 2–D blocking optimization method. All profiling information that is measured are stated as *samples*. The sampling rates defines how often a specified event should be measured. For default every 1024 periods a sample is counted. In the last line of the figure where the `RELAX` macro is found, one can see the sample rate for both events which are used for this instructions.

The `RELAX` macro computes one Gauss–Seidel iteration for every unknown as show in Equation 2.9. For a 5–point stencil there are ten floating–point (fp) operations needed: one fp–division, five fp–multiplication, three fp–summation and one fp–subtraction. Also for the 5–point–stencil the necessary amount of data must be fetched before the computation can be started. This includes the four neighbors for each unknowns, the according coefficients and the value for the right–hand–side.

For this reason it does not astonish that most of the cycles are spent in the `RELAX` macro. This is (besides the restriction and the prolongation) the only location inside the source code where fp–operations are made. For our computation example with one million unknowns ten million fp–operations need to be calculated. The size of data that must be fetched before each unknown can be iterated offers the possibility for the compiler to do loop unrolling. While the actual unknown is relaxed the data that is needed to smooth the next unknown can be prefetched using load operations. When the approximation for the actual unknown is done the data to computed the next point is already loaded into the registers. Because the Alpha 21164 has multiple independent units for fp–operations and for load instructions this can be done parallel using the multiple function units.

With the `dcpilist` tool one can additionally show the according machine code that was produced by the compiler. At the beginning in Section 4.5 the `gprof` tools shows where the most execution time is spent. In comparison to the `gprof` tool, the `dcpilist` tool also shows exactly which instructions

```
    0      2        ?        for (y = 1 ; y < yup; y=y+MG_BLOCK_y) {
    0     52        ?            for (x = 1 ; x < xup; x=x+MG_BLOCK_x){
    1    433        ?                for (block = 0 ; block < 2 * numberof_block ; block++){
    0      7        ?                    ybegin = y - block;
    0     35        ?                    yend   = ybegin + MG_BLOCK_y;
    0      0
    0      0                             /* ensure that square in y-direction is in grid */
    0      0                             if (ybegin < 1)
    1     37        ?                      ybegin = 1;
    0      0                             if (yend > ny)
    0    105        ?                      yend = ny;
    0   1285      7.5cy+?                for (yblock=ybegin; yblock < yend ; yblock++)
    0      0                               {
    0      0                                 xbegin= x - block;
    0     42        ?                        xend  = xbegin + MG_BLOCK_x;
    0      0
    0      0                                 /* ensure the square in x-direction is in grid */
    0      0                                 if (xbegin < 1)
    0    412      2.3cy+?                      xbegin = 1;
    0      0                                 if (xend > nx)
    0    169        ?                          xend = nx;
    0      0
    0      0                                 /* look if xbegin should start with a red or a black point
    0      0                                    if block = 0 (, 2, 4, 6, ... even)     then relax RED points
    0      0                                    if block = 1 (, 3, 5, 7, ....not even) then relax BLACK points */
    2   1244      7.9cy                      xbegin = xbegin + ( (xbegin + yblock + block) % 2);
    0    192      0.1cy                      for (xblock=xbegin; xblock < xend; xblock=xblock+2)
    0      0                                   {
    0      0                                     /* printf("u(%d, %d) ", xblock,yblock);*/
12414 360343    109.3cy                          RELAX(level, xblock, yblock); }}}}}
```

Figure 5.38: Output of the `dcpilist` tool for the `smooth` function for our multigrid program.

are responsible inside the function.

For the output in Figures 5.39 and 5.40 we compiled the same source code with the compiler optimizations in order to investigate whether the compiler performs loop unrolling or not. The output of the machine code also denounces the memory address in difference to the previous output where only the source code is shown.

In this figures one can see that the `RELAX` macro is split in a set of machine instructions. The output can be divided in five characteristic parts each beginning with a `divt` statement. Inside all of the five parts a complete Gauss–Seidel step is done. These instructions can be identified with the fp–division that have to be done with each Gauss–Seidel iteration. The `divt` instructions shows that `double` values are used for the instruction. Each fp–division is followed by nine other fp–operations that are needed to compute the rest of the Gauss–Seidel method. These are exactly the ten fp–operations that have to be computed for the complete algorithm. After the tenth fp–operation the result is stored. The `stt` instructions stores the new approximated result in the according grid point.

At machine address `0x120002f0c` the `beq` (= branch if equal) instruction can be found. The destination for this branch is located at `0x1200030c4` in the fifth part. If this jump is not executed the upper four parts are executed. For this we identify the upper four parts as the the unrolled loop for the x–direction.

At the end of the fourth part two branches can be found. If the first branch at address `0x1200030bc` is done the upper four parts are executed again. The loop ends when the jump is not done any more and the next branch instruction at `0x1200030c0` is true. Then the execution of the loop is finished.

With this perception one can see that the loop that executes the `RELAX` macro was unrolled four times. For that we can interprete the execution of the unrolled loop with the fact that the compiler has detected that the block which is used for our 2–D blocking algorithm, completely fits into the grid. For that the compiler unrolls the loop four times. If the block does not fit into the grid and has to be resized the fifth part without unrolling is executed.

Inside the machine code also some `ldt` instructions can be found[4]. For this we can see that the

[4] These loads are not shown in detail in the both figures.

```
12320 245849   267.9cy                             RELAX(level, xblock, yblock);
*   0      0     0.0cy        0x120002f0c beq     a5, 0x1200030c4

   28    945     1.0cy        0x120002f10 ldt     $f1, 8(s3)
 2236  17344    19.0cy        0x120002f44 divt    $f0,$f1,$f1
   30   1686     1.8cy        0x120002f48 mult    $f10,$f11,$f10
  ...
   79   2246     2.5cy        0x120002f5c mult    $f12,$f13,$f13
   15   3353     3.7cy        0x120002f60 addt    $f10,$f13,$f10
  182   4824     5.3cy        0x120002f70 mult    $f11,$f14,$f11
  ...
   99   5046     5.5cy        0x120002f80 mult    $f13,$f14,$f13
    0    287     0.3cy        0x120002f84 addt    $f10,$f11,$f10
  ...
    3   1936     2.1cy        0x120002f8c addt    $f10,$f13,$f10
   81   5346     5.8cy        0x120002f90 subt    $f10,$f11,$f10
    9   3277     3.6cy        0x120002f94 mult    $f1,$f10,$f1
   70   3599     3.9cy        0x120002f98 stt     $f1, -64(t12)
  -------------------------------------------------------------
 2349  22773    24.9cy        0x120002fb4 divt    $f0,$f14,$f14
    8   1011     1.1cy        0x120002fb8 mult    $f12,$f11,$f11
  ...
   83   2717     3.0cy        0x120002fc0 mult    $f13,$f10,$f10
    9   3176     3.5cy        0x120002fc4 addt    $f11,$f10,$f10
  ...
  110   3666     4.0cy        0x120002fd0 mult    $f1,$f12,$f1
  ...
   35   4955     5.4cy        0x120002fdc mult    $f11,$f12,$f11
    5   1150     1.3cy        0x120002fe0 addt    $f10,$f1,$f1
  ...
   11   2556     2.8cy        0x120002fe8 addt    $f1,$f11,$f1
   23   3606     3.9cy        0x120002fec subt    $f1,$f10,$f1
   22   3392     3.7cy        0x120002ff0 mult    $f14,$f1,$f1
   26   3479     3.8cy        0x120002ff4 stt     $f1, -48(t12)
  -------------------------------------------------------------
 2135  19679    21.5cy        0x120003010 divt    $f0,$f12,$f12
   10   1595     1.7cy        0x120003014 mult    $f13,$f10,$f10
  ...
   42   3208     3.5cy        0x120003020 mult    $f11,$f14,$f14
   27   3297     3.6cy        0x120003024 addt    $f10,$f14,$f10
  ...
   70   3005     3.3cy        0x120003030 mult    $f1,$f13,$f1
  ...
   88   4090     4.5cy        0x120003040 mult    $f14,$f13,$f13
    1    341     0.4cy        0x120003044 addt    $f10,$f1,$f1
  ...
   18   2393     2.6cy        0x12000304c addt    $f1,$f13,$f1
   77   4471     4.9cy        0x120003050 subt    $f1,$f10,$f1
   26   3403     3.7cy        0x120003054 mult    $f12,$f1,$f1
   78   3689     4.0cy        0x120003058 stt     $f1, -32(t12)
  -------------------------------------------------------------
 2425  23396    25.6cy        0x120003074 divt    $f0,$f14,$f14
    9   1218     1.3cy        0x120003078 mult    $f11,$f10,$f10
  ...
   61   2656     2.9cy        0x120003080 mult    $f13,$f12,$f12
  ...
    7   3341     3.7cy        0x120003090 addt    $f10,$f12,$f10
  ...
  240   7639     8.4cy        0x12000309c mult    $f13,$f12,$f12
   59   1483     1.6cy        0x1200030a0 mult    $f1,$f11,$f1
  ...
   25   2885     3.2cy        0x1200030a8 addt    $f10,$f1,$f1
   38   3457     3.8cy        0x1200030ac addt    $f1,$f12,$f1
   44   3592     3.9cy        0x1200030b0 subt    $f1,$f11,$f1
   23   3312     3.6cy        0x1200030b4 mult    $f14,$f1,$f1
   24   3397     3.7cy        0x1200030b8 stt     $f1, -16(t12)

*   0      0     0.0cy        0x1200030bc bne     a5, 0x120002f10
*  10   1041        ?         0x1200030c0 beq     s5, 0x120003148
  -------------------------------------------------------------
```

Figure 5.39: Output of the `dcpilist` tool for the `smooth` function. Also the machine code that is produced by the compiler is shown (part1).

```
     0    31      ?      0x1200030c4 ldt     $f10, 8(s3)
    31   263      ?      0x120003100 divt    $f0,$f10,$f10
     0    15      ?      0x120003104 mult    $f13,$f12,$f12
     0     0             0x12000310c ldq_u   zero, 0(sp)
     0    43      ?      0x120003110 mult    $f11,$f14,$f11
     0     1      ?      0x120003114 ldt     $f14, -16(s1)
     1    73      ?      0x120003118 addt    $f12,$f11,$f11
     0     0             0x12000311c ldt     $f12, -56(s3)
     0    89      ?      0x120003120 mult    $f14,$f12,$f12
     0    20      ?      0x120003124 mult    $f1,$f13,$f1
     0     0             0x120003128 ldt     $f13, -96(s3)
     2    68      ?      0x12000312c addt    $f11,$f1,$f1
     0    84      ?      0x120003130 addt    $f1,$f12,$f1
     1    84      ?      0x120003134 subt    $f1,$f13,$f1
     2    57      ?      0x120003138 mult    $f10,$f1,$f1
     0     0             0x12000313c ldq_u   zero, 0(sp)
     1    85      ?      0x120003140 stt     $f1, -16(t12)
*    0     0             0x120003144 bne     s5, 0x1200030c4
```

Figure 5.40: Output of the `dcpilist` tool for the `smooth` function. Also the machine code that is produced by the compiler is shown (part2).

compiler also uses *outstanding loads*. This optimization is a kind of prefetching. The compiler loads the next values needed for further computation into the registers in order that they are ready to use in the following instructions. With this methods a lot of stalls due to load instruction can be prevented.

In the next we made an investigation whether the compiler benefits if the source code already is unrolled. For this we used version 4 of our optimization method. In this optimization method grid is divided in different regions in order to help the compiler using unrolling. For that we expanded the source code in the following way: in version 5 we explicite unrolled the inner loop (the block size in x–direction) in the middle interior part. In version 6 we additionally unrolled the block size to the y–direction.

The performance measurements also lead to no significant performance gain than for the original version. The profiling additionally also shows no difference for any of the both unrolled version in comparison to version 4. With explicite unrolling this 2–D blocking method was also slower than version 3 of the 2–D blocking method. Apparently in this blocking method the compiler can perforce unrolling more efficient than for the other versions. Maybe the additional computation effort that must be made for version 4 leads to this performance decrease.

For the loop unrolling method that the compiler made we must also note that the Alpha 21164 compiler for our multigrid problem always tried to unroll the inner loop — the loop in the x–direction — four times if no other compiler option is specified. For this reason if the block in x–direction is set smaller (so only two red or black operations can be unrolled) the compiler produces the same source code and tries to unroll four times. This unrolling optimization does not make sense. Therefore if this code is executed the profiling tool shows that the unrolling part of the machine code was not executed.

For our multigrid problem we also had to claim that with these small block sizes in x–direction 2–D blocking did not make sense in terms of cache usage and data locality. But on the other hand it shows that the loop unrolling which is made by the compiler is not always done the best and efficient way. For this example the number of unrolled loops must be stated explicite to the compiler with the `-unroll 2`).

For version 3 of our 2–D blocking algorithm we can also see that the most stalls that occur can be reduced to the five fp–division that is done with `divt` instruction. These results can be seen using the `dcpitopstalls` tool, shown in Figure 5.41. In the figure we can find the number of cycles for the according stall. The according instruction can be found comparing the address with the previous profiling data; already is already shown in Figure 5.39 and 5.40. This leads to the fact that the five uppest stalls in the figure again belongs to `divt` instruction. The conclusion for this is that theses stalls almost could not be improved because when the `divt` instruction has to be executed it consumes a lot of (static) stalls that are needed to perform this computation. In contrast to multigrid methods using constant coefficients, this division has be computed only once, because the coefficients do not change. For variable coefficients the fp–division must be computed for every grid point that should be relaxed.

At the end of this section we have compared our first unoptimized version with the best optimized version for the multigrid algorithm with variable coefficients. In order to show the number of stalls that arise we used the `dcpiwhatcg` tool. With it the number of static and dynamic stalls are shown in percentage. The number of stalls that arise can be determined with the `dcpitopstalls` tool.

In the below table one can see the number of stalls within the optimized and best version of our multigrid program compared with the number of stalls that arise with the unoptimized version. The values are taken using the `dcpitopstalls` and the `dcpiwhatcg` tool. The output for the `dcpiwhatcg` program can be found in Figure 5.42 for the optimized version and the unoptimized version is shown in Figure 5.43.

multigrid version	*number of stalls*	*static stalls (in %)*	*dynamic stalls (in%)*
optimized version	197949	25.5%	52.1%
unoptimized version	1208051	2.8%	92.3%

We have shown that the number of stalls during our optimization phase reduces about a factor of 10. The number of dynamic stalls reduces of about one million samples of about 100.000 samples. This is also an effect of the fewer cache misses and the better usage of local data. With the loop unrolling effects also multiple iterations can be done by parallel in comparison to the unoptimized version. The number of multiple functional units can be used more effectively. When the data is more local also fewer dynamic stalls arise and with it the performance of our program increases. The top stalls that arise for the optimized version are fp–divisions that of course could have to be computed for multigrid methods using variable coefficients. For our unoptimized version the tops stalls that arise in comparison are load operations that are eliminated using the cache optimization methods.

top 100 stalls of length ≥ 0:									
%	cum%	cycles	count	C	avg	blame	pc	procedure	file:line
8.9%	8.9%	22939	914	M	25.1	df	0x120003074	mgSmooth	mg-func.c:783
8.6%	17.5%	22316	914	M	24.4	df	0x120002fb4	mgSmooth	mg-func.c:783
7.4%	24.9%	19222	914	M	21.0	df	0x120003010	mgSmooth	mg-func.c:783
6.5%	31.4%	16887	914	M	18.5	df	0x120002f44	mgSmooth	mg-func.c:783
2.2%	33.6%	5651	914	M	6.2	d	0x12000309c	mgSmooth	mg-func.c:783
1.8%	35.3%	4589	914	M	5.0	id	0x120002f80	mgSmooth	mg-func.c:783
1.4%	36.7%	3633	914	M	4.0	id	0x120003040	mgSmooth	mg-func.c:783
1.3%	38.1%	3381	914	M	3.7	d	0x120002f70	mgSmooth	mg-func.c:783
1.2%	39.3%	3199	914	M	3.5	a	0x120002f90	mgSmooth	mg-func.c:783
1.2%	40.5%	3199	914	M	3.5	a	0x120003050	mgSmooth	mg-func.c:783

Figure 5.41: Output of the `dcpitopstalls` tool for the `smooth` function.

Where have all the cycles gone?				
I-cache (not ITB)	0.3%	to	6.6%	
ITB/I-cache miss	0.0%	to	0.0%	
D-cache miss	9.6%	to	47.2%	
DTB miss	3.4%	to	4.4%	
Write buffer	0.0%	to	0.6%	
Synchronization	0.0%	to	0.0%	
Branch mispredict	0.0%	to	0.2%	
IMUL busy	0.0%	to	0.0%	
FDIV busy	0.0%	to	31.5%	
Other	0.0%	to	0.0%	
Unexplained stall	0.4%	to	0.4%	
Unexplained gain	-0.3%	to	-0.3%	
Subtotal dynamic				52.1%
Slotting	0.6%			
Ra dependency	17.8%			
Rb dependency	5.8%			
Rc dependency	0.0%			
FU dependency	1.4%			
Subtotal static				25.5%
Total stall				77.6%
Useful	20.6%			
Nops	1.8%			
Total execution				22.4%

Figure 5.42: Part of the output that `dcpiwhatcg` shows with the next optimization method used within our multigrid program.

Where have all the cycles gone?				
I-cache (not ITB)	0.0%	to	38.2%	
ITB/I-cache miss	0.0%	to	0.0%	
D-cache miss	18.8%	to	43.5%	
DTB miss	30.9%	to	49.4%	
Write buffer	0.0%	to	0.0%	
Synchronization	0.0%	to	0.0%	
Branch mispredict	0.0%	to	0.0%	
IMUL busy	0.0%	to	0.0%	
FDIV busy	0.0%	to	8.1%	
Other	0.0%	to	0.0%	
Unexplained stall	0.0%	to	0.0%	
Unexplained gain	-0.7%	to	-0.7%	
Subtotal dynamic				92.3%
Slotting	0.0%			
Ra dependency	2.7%			
Rb dependency	0.0%			
Rc dependency	0.0%			
FU dependency	0.1%			
Subtotal static				2.8%
Total stall				95.1%
Useful	4.1%			
Nops	0.8%			
Total execution				4.9%

Figure 5.43: The number of stalls with the `dcpiwhatcg` tool used with the unoptimized version of our program.

Chapter 6

Conclusions

In this thesis we have concentrated on the optimization of 2–D multigrid algorithms with variable coefficients. Therefore we have implemented a 2–D multigrid algorithm on structured rectangular grids and Dirichlet boundary conditions. The multigrid program uses a V–cycle scheme to compute the error corrections on the coarser grids. If the coarsest grid contains a single unknown, it can be solved using one Gauss–Seidel step. In order that the residual does not need to be restricted until the coarsest grid only consists one unknown we additionally have implemented a direct solver to compute the solution on the coarsest grid. The direct solver is implemented using the *LAPACK* library, which makes our multigrid program more flexible.

The optimization methods are not dedicated to specific architectures but rather use general considerations about optimization methods to improve data locality. In order to show how the optimizations improve the performance of the multigrid program we used different architectures for the measurements. For each architecture that is used inside this thesis a brief overview of its design — with a special focus on its memory structure — is provided. The cache misses that occurred are measured using the *PCL* library which reads the performance counters for various architectures. The profiling of our program is done using the *DCPI* tool that only works on the Alpha–based architectures.

In our optimization method we showed that the `smooth()` function consumes most of the execution time. Therefore the optimization concentrates on this function. First we have shown that also for basic methods like *loop interchange* — which are easy to implement but often ignored — enormous speed improvements could be received. Afterwards, our main attention for the optimization methods has been paid to data layouts and blocking methods.

The data layout covered strategies in which way the data structures have to be arranged in order to improve data locality. Combined with this we demonstrated *padding* strategies and showed their influence with the resulting speedups. The performance therefore highly depends on the occurrence of *thrashing effects* within the data structure. Particularly we showed that with a merged data layout — that is more insensitive towards these thrashing effects — usually a better performance can be reached.

For the blocking optimizations we started implementing 1–D blocking methods with multiple lines blocked. For this method we observed that the speed increase highly depends on the number of blocked iterations and on the size of a grid line. If multiple grid lines cannot be held in cache the speedup of this method also suffers from arising cache misses. This effect could well be noted for architectures with small caches. The reason for this is the growing working set which does not fit into the cache any more.

The next step led to 2–D blocking methods. With these methods the block could be made variable in size which should lead to a more cache efficient usage of the data. The measurements verified these considerations. On the other hand growing speed improvements could only be seen if multiple iterations of the smoother are blocked. The reason for this is the growing computational overhead that has to be made in comparison to the 1–D blocking methods. Also the size of the block leads to different speedups. With this the influence that an improved data locality has is shown. We again received the best performance if the working set completely fits into the cache. Therefore the block sizes have to be chosen with respect to the used architecture and their cache sizes to receive the best performance.

In the following table we show the results of our speed measurements for various architectures.

	Alpha 21264 (XP 1000)		Alpha 21164 (500 au)		Pentium II (350)	
	opt.	unopt.	opt.	unopt.	opt.	unopt.
total time	10.6	26.0	23.9	70.1	36.5	61.1
smoothing time	7.8	21.9	17.0	60.0	28.7	52.0
restriction time	2.3	2.0	4.6	5.1	5.3	4.7
prolongation time	0.2	1.7	1.2	4.5	1.5	3.7
L1 cache misses	—	—	—	—	$2.5 * 10^7$	$6.2 * 10^7$
L2 cache misses	—	—	$1.5 * 10^7$	$5.2 * 10^7$	$9.6 * 10^7$	$3.8 * 10^7$

For this we measured the performance for the unoptimized version of the multigrid method in comparison to the best results for our optimizations. For the measurements a grid with about one million unknowns is used with ten V–cycles and four pre– and post–smoothing steps each. For this the problem size can be seen as representative for high performance computing. The amount of data for this problem size is about 56 MB. We restricted until the coarsest grid contains one unknown which means that the solution can be computed directly with one Gauss–Seidel iteration.

In the results one can see that the execution time for the `smooth()` function of the multigrid algorithm was reduced drastically. For this also the total time for the algorithm is decreased. The best results could be found for both Alpha architectures. There the speed increase was about three times faster than with the unoptimized version. For the Intel Pentium II the speed increase was only about 70%.

For the improvement of the cache misses almost the same factor than for the speed increase could be found. This verifies the assertion that in today's CPUs the main limiting factor is the cost of the memory references. The MFLOPS results are far away form the *peak performance* that these CPUs can reach theoretically. For the results in the table we computed about $1.4 * 10^8$ floating point operations. This leads to a MFLOPS rate about 130 MFLOPS for the Alpha 21264 which has a theoretical peak performance of about 1 GFLOPS. On the other hand one must keep in mind that we computed a problem with about one million unknowns, leading to a data set size of 56 MB.

We also showed on the basis of a case study where most of the performance loss is localized in the machine code. The numbers of top stalls of our optimized program in comparison to the unoptimized version are shown. For this one could see that with the optimized version the number of *dynamic* stalls decreases significantly in contrast to the number of *static* stalls which is also an indication for better data locality.

For further algorithmic extensions that can be made to our multigrid program one can think of other boundary conditions than the implemented Dirichlet boundaries. Also the choice of the coarse grid coefficients can be handled in another way if one e.g. thinks of Galerkin products.

Finally if the results are summarized one can see that the measured MFLOPS rates are not very high in comparison to the provided peak performances that the different CPUs have. On the other hand one must keep in mind that with variable coefficients the amount of data highly increases in contrast to constant coefficients. Nevertheless the optimization methods improved the data locality leading to a more cache–efficient usage. For this we gain a high performance improve in contrast to the unoptimized version. With the results in this thesis we have shown that the performance of a program can highly be influenced with a more cache–efficient implementation.

Appendix A

Program Parameters

In this chapter we describe all program parameters which can be passed to our multigrid program. These parameters are used to influence the algorithm, the grid size and the measurements.

In the beginning we describe the multigrid parameters shown in Table A.1. With these parameters the grid size, the number of V–cycles, the number of pre– and post–smoothing step and the number of grids can be defined. The next parameter refer to the included PCL function calls. Thus the event type which is to measure defined in the PCL header file can be specified.

As described in Chapter 4 the grid is initialized as a square. Only the number of intervals need to be parsed. The `-nints` parameter sets the number of intervals for each dimension.

Because the algorithm performs no residual test and breaks if it reaches a defined accuracy the number of V–cycles has to be passed. This can be done with the `-iter` option.

The number of pre– and post–smoothing steps can be influenced with the `-v1` and `-v2` parameter.

The number of grids can be specified with the `-ngrids` option. When this parameter is given the coarsest grid is computed directly using the *LAPACK* library. This is important for further improvements that can be made. The finest grid is not limited to a square. The number of intervals in x– and y–direction can be of different size. But for this also the initialize function must be adapted.

The `-event` parameter influences the PCL measurements. We provided up to three events that can be measured simultaneously. The number of events which can be measured depends on the number of performance counters for the used architecture and the PCL implementation. For some architectures even more events can be measured in parallel.

For the time measurements no parameters are needed. In our program implementation we measured the total time, the smooth time, the prolongation time and the time that was used for the restriction.

All parameters that were described above are written to a logfile. With this file the measurement of our performance results are verified.

parameter	function
`-nints`	The number of intervals the grid has in each direction. With this the x– and y–direction are set up. The grid therfore includes $(\text{nints} + 1)^2$
`-iter`	The number of V–cycles the algorithm is started

`-v1`
 	
`-v2`
	The pre–smoothing steps that are made with the the red–black
	Gauss–Seidel smoother
	The number of post–smoothing steps

`-ngrids`	The number of grids that should be used. The coarsest grid is solved directly using the *LAPACK* library.
`-event{1,2,3}`	Number of *PCL*–events that can be measured
`-logfile`	Name of the logfile. This file includes all the described parameters above (multigrid data, kind of measurement and measurement data

Figure A.1: The parameters used to configure the multigrid algorithm.

Appendix B

Optimization variants

In this chapter we describe in which way the several optimizations were integrated in our program. Every optimization variable that is explained here is implemented using preprocessor directives. This means that the different variables must be set at compile time. Table B.1 shows the different variables.

In the beginning the different optimization variants are explained. After that the table shows how the data layouts are defined. Then several optimizations flags, such as the padding size that corresponds to the different data structures, the size of the block in x– and y–direction and the number of blocked iterations is explained for the 1–D and 2–D blocking. Finally some additionally variables were described which are need if e.g. different grid initialization are used or the grid is to be smoothed only on the the finest grid and if the results for every V–cycle should be written to a file.

The main optimization that is done in our multigrid program claims the `smooth()` function. We developed several functions that compute the red–black Gauss–Seidel algorithm on different ways. Therefore we started with an unoptimized version until we implemented several 2–D blocking methods at the end. In order to compare the performance between the different optimization methods we left the several `smooth()` function inside our source code. With this one can reconstruct the prior optimization levels. In the last two optimization steps — labeled version 5 and version 6 — we additionally unrolled the `RELAX()` macro which iterates each grid point in order to show how efficient the compiler works. The kind of *optimization* can be set using the `-DOPT` switch.

The data layout is specified with the `-DDATA` switch. The different data structures are explained in detail in Section 5.2. For each data structure a different kind of padding is defined. Therefore with `-DMG_PAD_1` and `-DMG_PAD_2` the size of the padding variables can be set.

The `-DMG_BLOCK` defines the number of iterations that are blocked with one sweep through the grid. This variable is only needed when blocking — 1–D blocking or 2–D blocking — is used.

The both `-DMG_BLOCK_x` and `-DMG_BLOCK_y` determine the size of the block for the 2–D blocking optimization method. For our multigrid algorithm we decided to used only even block sizes in x– and y–direction. This makes it easier to differentiate between the red and the black points inside the block. Also this restriction should have no impact on the performance measurements.

Because the multigrid algorithm lacks of general initialization function we additional implemented three initialization routines. With the `-DPN` switch a standard example problem that was also used for our performance measurements is computed. Therefore we initialized $u = 1$ on G and $f = 0$. The Dirichlet boundary condition is set as: $u = 0$ on ∂G. If `-DPSIN` is defined we initialized the solution vector with two overlapping sinus functions. The cause for this was to demonstrate the smoothing properties for the Gauss–Seidel algorithm. This example is used to demonstrate how the red–black Gauss–Seidel algorithm eliminates the high frequent error components. In difference to this the error correction that is solved on the coarser level is used to eliminate the low frequent error parts. The last initialization is used to show that with variable coefficients much more flexible problems can be solved. The problem is described in [Rüd81]. With this problem initialization one

parameter	function
-DOPT_{x}	several optimization variants that were applied to the `smooth()` function 0: no optimization 1: loop interchange 2: loop fusion 30: 1–D blocking (1 iteration blocked) 31: 1–D blocking (multiple iterations can be blocked) 40: 2–D blocking (1. version) 41: 2–D blocking (2. version) 50: 2–D blocking (3. version, multiple iterations can be blocked) 51: 2–D blocking (4. version, multiple iterations can be blocked, borders were separated) 52: 2–D blocking (5. version, normal loop unrolling) 53: 2–D blocking (6. version, extended loop unrolling)
-DDATA_{x}	number of the data layout that should be used; {1,2}: unoptimized data structures {3,5,6}: data is stored band wise {4,7,8}: data merging storage
-DMG_PAD_1 -DMG_PAD_2	These both variables define the used padding within the data layout. Both padding variables should be initialized with a variable ≥ 0. The padding method depends on the selected data structure.
-DMG_BLOCK_x -DMG_BLOCK_y -DMG_BLOCK	Block size in x–direction Block size in y–direction Number of blocked iterations
-DP{N, SIN, PTOS}	Different initialization of the starting grid and the coefficients. N: $\vec{u}_h = \vec{1}$ and $\vec{f}_h = \vec{0}$ SIN: $\vec{u}_h = \mathtt{sin()sin()}$ and $\vec{f}_h = \vec{0}$ TOS: treatment of singularities
-DSMOOTH	Do iteration only on the finest level. No coarse level correction is performed.
-DMTIME -DMPCL	Additional functions are called for time measurements. Additional functions are called for *PCL* measurements.
-DLOG	Write a logfile were all variables and attributes about the multigrid algorithm (i.e. the grid size) and also the measured results for the performance comparison were logged to.
-DPLOT	Plot the results of the both vectors $\vec{f}$ and $\vec{u}$ for each grid. The results are written to separate files.
-DFP_COUNT	Count the number of floating–points operations.

Figure B.1: Several parameters that should be set at compile time.

can show the behavior of the algorithm with strong singularities between the coefficients.

These different initialization methods do not influence the performance of the multigrid program, because for each kind of problem the same amount of data needs to be stored. Also the cost for each computation is the same for every problem definition given. The different problems are only used to show the difference between multigrid methods and iterative solvers within this thesis.

In order to measure the speed of the `smooth` function with the `-DSMOOTH` statement only the finest grid is smoothed. With this only the performance for the smoother is measured.

`-DMTIME` and `-DMPCL` define which performance measurement should be used. If `-DMPCL` is used we call the *PCL* function inside our program. The output can be written to a logfile. With the `-DMTIME` time measurements are done using the *ANSI–C* `clock()` function that measures the processor time used by the program.

With the `-DLOG` switch all important multigrid variables are logged to a file. The file consists about multiple lines. In every line the following values are logged: the number of V–cycles, the number of intervals of the grid, the number pre– and post–smoothing steps. After that also the defines about the kind of problem defined — using one of these switches `-DP{N, SIN, PTOS}` —, the kind of optimization, the blocking variables and the data layout is written. In the end of the line the performance results are logged.

The `-DPLOT` writes an output for the right hand side and the solution vector on the different grids for every V–cycle. At every grid the solution vector is written after each pre– and post–smoothing step. The right hand side, which transfers the error correction between two grids is also written. In these files the results of the vectors are arranged for the the use with the *gnuplot* tool. With this tool all the outputs that the multigrid program delivers are plotted. For further information see [Cen99].

For architectures where *PCL* is not implemented the number of floating–points using the `-DFP_COUNT` statement are computed by our program. With this every floating–point operation within a V-cycle is counted to calculate the MFLOPS rate.

Bibliography

[AMD99] AMD. *AMD Athlon Processor Technical Brief.* http://www.amd.com/products/cpg/athlon/techdocs/pdf/22054.pdf, December 1999. Publication #22054.

[Ber00] R. Berrendorf. *PCL – The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors (Version 2.0).* Forschungszentrum Juelich GmbH, Federal Republic of Germany, http://www.fz-juelich.de/zam/PCL/, September 2000.

[Bra77] A. Brandt. Multi Level Adaptive Solutions to Boundary–Value Problems. *Mathematics of Computation*, 31, 1977.

[Bra84] A. Brandt. Multigrid Techniques: 1984 Guide with Applications to Fluid Dynamics. *GMD Studien*, 85, 1984.

[Bri00] W. L. Briggs. *A Multigrid Tutorial.* SIAM, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, second edition, 2000.

[Cen99] Gnuplot Central. *gnuplot.* http://www.gnuplot.org, November 1999.

[Com96] Compaq. *Alpha 21164 Microprocessor Hardware Reference Manual.* http://www.support.compaq.com/alpha-tools/documentation/archive/21164/ec-qaeqd-te_21164_hrm.pdf, July 1996.

[Com99] Compaq. *Technology for Performance: Compaq Professional Workstation XP1000.* http://www5.compaq.com/support/techpubs/whitepapers/ECG0500199.html, January 1999. ECG050/0199.

[Com00] Compaq. *Compaq (Digital) Continuous Profiling Infrastructure (DCPI).* http://www.tru64unix.compaq.com/dcpi/, November 2000.

[DS99] K. Dowd and C. Severance. *High Performance Computing.* O'Reilly and Associates, Inc., 101 Morris Street, Sebastopol, CA 95472, 1999.

[gpr98] *GNU gprof.* http://www.gnu.org/manual/gprof-2.9.1/, November 1998.

[Hac85] W. Hackbusch. *Multigrid Methods and Applications.* Springer Verlag, Berlin, 1985.

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

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture a Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, California, second edition, 1996.

[Int99a] Intel. *Intel Architecture Software Developers Manual, Volume 1: Basic Architecture.* http://developer.intel.com/design/PentiumII/manuals/243190.htm, 1999.

[Int99b] Intel. *Intel Architecture Software Developers Manual, Volume 3: System Programming Guide.* http://developer.intel.com/design/PentiumII/manuals/243192.htm, 1999.

[lap99] *LAPACK Users' Guide.* `http://www.netlib.org/lapack/lug/lapack_lug.html`, third edition, August 1999.

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

[LRM00] Leibniz Rechenzentrum München. *Höchstleistungsrechner in Bayern (HLRB): The Hitachi SR8000-F1.* `http://www.lrz-muenchen.de/services/compute/hlrb/`, 2000.

[MG80] A. Mitchell and D. Griffiths. *The Finite Difference Method in Partial Differential Equations.* John Wiley & Sons Ltd., Chister, 1980.

[Pac98] Hewlett Packard. *PA–8500: The Continuing Evolution of the PA–8000 Family.* `http://www.ia-64.hp.com/news_events/archives/pa8500_print.html`, 1998.

[Rüd81] U. Rüde. The Multi–Grid Method for the Diffusion Equations with Strongly Discontinous Coefficients. *SIAM, Society for Industrial and Applied Mathematics*, 2, December 1981.

[Rüd97] U. Rüde. Iterative Algorithms on High Performance Architectures. In *Proceedings of the EuroPar97 Conference*, Lecture Notes in Computer Science, pages 26–29. Springer, August 1997.

[Sch97] H. R. Schwarz. *Numerische Mathematik.* B. G. Teubner, Stuttgart, fourth edition, 1997.

[ST82] K. Stüben and U. Trottenberg. *Multigrid Methods: Fundamental Algorithms, Model Problem Analysis and Applications*, volume 960 of *Lecture Notes in Mathematics.* Springer Verlag, Berlin, Heidelberg, New York, 1982.

[TCM94] C.-W. Tseng, S. Carr, and K. S. McKinley. Compiler Optimizations for Improving Data Locality. In *Sixth International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS-VI)*, San Jose, Oktober 1994.

[TR98] C.-W. Tseng and G. Rivera. Data Transformation for Eliminating Conflict Misses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[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.