Data Locality Optimizations for Multigrid Methods on Structured Grids

Christian Weiß

Technische Universität München
Institut für Informatik
Lehrstuhl für Rechnertechnik und
Rechnerorganisation

Data Locality Optimizations for Multigrid Methods on Structured Grids

Christian Weiß

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Hans Michael Gerndt

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Arndt Bode
2. Univ.-Prof. Dr. Ulrich Rüde
   Friedrich-Alexander-Universität Erlangen-Nürnberg
3. Univ.-Prof. (komm.) Dr.-Ing. Eike Jessen, em.

Die Dissertation wurde am 26. September 2001 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 06. Dezember 2001 angenommen.

Abstract

Beside traditional direct solvers iterative methods offer an efficient alternative for the solution of systems of linear equations which arise in the solution of partial differential equations (PDEs). Among them, multigrid algorithms belong to the most efficient methods based on the number of operations required to achieve a good approximation of the solution. The relevance of the number of arithmetic operations performed by an application as a metric for the complexity of an algorithm wanes since the performance of modern computing systems nowadays is limited by memory latency and bandwidth. Consequently, almost all computer manufacturers nowadays equip their computers with cache–based hierarchical memory systems. Thus, the efficiency of multigrid methods is rather determined by good data locality, i.e. good utilization of data caches, than by the number of arithmetic operations.

In this thesis, the cache and memory access behavior of multigrid methods is systematically analyzed for the first time. The analysis is based on an exhaustive study of modern microprocessor memory hierarchies. Detailed runtime as well as theoretical studies of the performance of these methods demonstrate the interaction between multigrid algorithms and deep memory hierarchies. In particular, issues involved with the multilevel nature of the memory hierarchy are addressed. Furthermore, delays due to main memory accesses are clearly revealed as the performance bottlenecks of multigrid methods and their components. Besides the performance bottlenecks, upper limits for the achievable performance of multigrid methods on RISC based microprocessors are determined by means of theoretical models.

Based on the knowledge gained from the analysis of multigrid algorithms and microprocessor architectures, new data locality optimization techniques for multigrid methods are proposed. The techniques extend existing code and data layout restructuring techniques and are able to significantly improve data locality and consequently speed up the execution of multigrid algorithms by a multiple. With the improved data locality multigrid methods are able to utilize 15 to 30 per cent of the peak performance on a multitude of modern computer systems. The impact of the techniques is demonstrated with runtime and memory hierarchy behavior measurements as well as theoretical data locality examinations.

The applicability of the techniques is demonstrated by means of the DiMEPACK library. DiMEPACK is a multigrid solver for two–dimensional problems with constant coefficients on structured grids. In this thesis, however, aspects of multigrid methods for three–dimensional problems and variable coefficients are discussed as well.

Acknowledgments

First of all, I would like to thank my doctoral adviser, Prof. Dr. Arndt Bode, for his valuable support throughout the years. He provided an excellent research environment and great freedom to carry out my research.

Furthermore, I would like to thank my co–adviser Prof. Dr. Ulrich Rüde and Prof. Dr. Hermann Hellwagner for their technical advice. Despite their positions they proved to be committed colleagues and it was a pleasure to work with them on the DiME project.

I would also like to thank Prof. Dr. Eike Jessen for taking the duty to be my second co–referee in an unbureaucratic way and for providing valuable hints.

A special thank–you goes to Dr. Wolfgang Karl, Markus Kowarschik, and Oliver Creighton. They spent many hours of their spare time to read my thesis and provide numerous comments and ideas on how to improve this thesis.

The German Science Foundation (Deutsche Forschungsgemeinschaft) deserves credit for supporting my work on the DiME Projekt (research grants Ru 422/7–1,2,3).

I would also like to thank all of my former and current colleagues, in particular Georg Acher, Rainer Buchty, Michael Eberl, and Detlef Fliegl. Without them, work would have been a lonesome task.

Finally, I would like to thank my love Veronika for her support and patience.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

The performance of microprocessors increased significantly over the years so that nowadays standard computer systems provide a theoretical peak performance of one GFLOPS[1] and beyond. Nevertheless, it is a well–known fact that the speed of processors has been increasing much faster than the speed of main memory components. Recently, there is much effort in improving memory technology but many computer architects are expecting that the tendency will persist for at least a decade. As a general consequence, current memory chips based on DRAM technology cannot provide the data to the CPUs as fast as necessary. This memory bottleneck often results in significant idle periods of the processors and thus in very poor code performance compared to the theoretically available peak performance of current machines.

To mitigate this effect, modern computer architectures use multilevel cache memories which store data frequently used by the CPU. Caches are usually integrated into the CPU or based on SRAM chips. Both approaches deliver data much faster than DRAM components, but on the other hand have comparatively small capacities, for both technical and economical reasons. Efficient execution can therefore be achieved only if the hierarchical structure of the memory subsystem (including main memory, caches and the processor registers) is respected by the code, especially by the order of memory accesses. Unfortunately, even modern compilers are not very successful in performing data locality optimizations to enhance cache efficiency. As a consequence, most of this effort is left to the programmer.

Applications such as simulations of physical phenomena require apparently unlimited calculation power. Therefore, they are predestinated for more powerful microprocessors. These phenomena, for instance currents in fluids, are often described by partial differential equations (PDEs) which can be approximated by sparse systems of algebraic equations. To represent realistic two– resp. three–dimensional problems, however, systems of linear

[1]One MFLOPS $\equiv 10^6$ floating point operations per second. One GFLOPS $\equiv 10^9$ floating point operations per second.

equations with up to several hundred millions of unknowns are required. The data structures which store the equations easily consume several hundred Mbyte or even Gbyte of main memory.

Multigrid methods are among the most efficient algorithms for the solution of large systems of linear equations arising in the context of numerical PDE solution, based on the number of operations required to achieve a good approximation of the solution. Generally speaking, multigrid methods approximate the physical domain with one fine grid and several coarser grids. The problem on the finest grid is solved approximately with a smoother resp. relaxation method such as the red–black Gauss–Seidel algorithm. The coarser grids are in turn used to calculate corrections to the approximation. Information between grid levels is transferred with inter–grid transfer operations such as restriction and interpolation.

Multigrid algorithms belong to the class of iterative methods, which means that the underlying data set is repeatedly processed several times. Thus, during one iteration the whole or at least a large part of the data has to be delivered from main memory to the CPU to be processed. In the following iteration the same process happens again with a slightly changed data set. The fact that multigrid methods repeatedly access their data set promises high data locality, however, in reality this data locality is not exploited. The direct consequence of their bad data locality is that multigrid methods merely reach a small fraction of the available peak performance of current microprocessors.

The discrepancy between inherent and exploited data locality of multigrid algorithms forms the key motivation for this work. With a consolidated knowledge of microprocessor architectures with deep memory hierarchies and of the algorithmic behavior of multigrid methods it should be possible to restructure multigrid methods so that the inherently available data locality is exploited and consequently a better performance is achieved. It must be pointed out that the optimizations should not change the semantics of the multigrid algorithms so that standard multigrid convergence estimates apply to the new cache optimized multigrid codes.

Another approach to improve the performance of iterative methods which is not investigated in this thesis is *parallelization*. Many researches have implemented hand–coded efficient parallel multigrid methods very often based on domain decomposition techniques. While parallelization often leads to significant speedups, the computation on each node will still suffer from high latency and low bandwidth involved with main memory accesses. Thus, the work in this thesis will apply to both parallel and sequential multigrid codes. However, details and extensions to parallel codes are left for further research.

1.2 Contributions

This thesis reveals that the major problem, which prohibits the exploitation of the data locality inherently present in multigrid methods, is the fact that the data sets involved with realistic problems are much larger than the cache levels built in currently available or forecasted computer systems. The direct consequence of the bad data locality is a bad runtime performance on architectures with deep memory hierarchies. Based on a detailed study of multigrid methods and microprocessor architectures new data locality optimizations are proposed which significantly improve the runtime performance of multigrid methods. The key contributions of the thesis are as follows:

- For the first time a detailed and systematic study of the runtime, cache, and memory behavior of multigrid methods is performed. The analysis demonstrates the interaction between multigrid methods and deep memory hierarchies. The analysis is based on runtime measurements as well as on theoretical studies of the data locality and performance of multigrid methods. In particular, issues involved with the multilevel nature of the memory hierarchy are addressed. The analysis exposes that the performance of multigrid algorithms is determined by main memory accesses. When executing the smoother which is by far the most time consuming component of the multigrid method on a Compaq PWS 500au, for example, the microprocessor is idle 80 per cent of all cycles waiting for data to arrive from main memory. Furthermore, properties of multigrid methods are exposed which are responsible for the bad performance.

- Upper limits for the achievable performance of multigrid methods on RISC based microprocessors are determined by means of theoretical models. The models testify that multigrid algorithms will at best achieve 50 per cent of the available peak performance on typical microprocessors.

- New data locality optimization techniques for multigrid methods on structured grids are developed based on the knowledge gained from the detailed analysis of multigrid methods, modern microprocessor memory hierarchies, and their interaction. The techniques extend existing data locality optimization techniques such as *loop fusion* or *loop blocking* and are able to significantly improve the data locality of multigrid algorithms. The optimizations which focus on the smoother component are able to speed up the execution on currently available workstations and PCs by a factor of up to five, especially for large grid sizes. The significance of all optimization techniques is demonstrated by means of detailed performance analysis and theoretical discussions of data locality properties.

- Although the thesis focuses on optimization techniques for the smoother component, other optimization techniques for inter–grid transfer operations are proposed. These optimization techniques combine pre– and post–coarse–grid operations. Thus, the number of global sweeps through the data structure is reduced which in turn improves data locality and performance.

The applicability of the techniques will be demonstrated by means of the DiMEPACK library. DiMEPACK is a fully functional multigrid library for two–dimensional constant coefficient problems on structured grids. The fast smoothers and inter–grid transfer optimizations developed in this thesis establish the core elements of the DiMEPACK multigrid solver. The library demonstrated its robustness and efficiency in a global cell placement code for VLSI circuits developed at the Institute of Electronic Design Automation (LEA), Technische Universität München.

In addition to the major contributions, new cache visualization techniques are proposed based on the experience gained during the performance analysis. The new visualization techniques provide a more intuitive understanding of the dynamical nature of memory hierarchies. Some of the techniques have been integrated in the experimental memory hierarchy visualization tool called *MHVT*.

1.3 Thesis Organization

The thesis is structured as follows: Chapter 2 describes the architectural development and performance issues involved with high main memory latency and insufficient memory bandwidth. Common properties of all memory hierarchies are identified and establish the fundamentals for the algorithmic studies. In Chapter 3, the idea of multigrid methods is introduced and the runtime, cache, and memory behavior of multigrid algorithms is analyzed. Thus, the algorithmic properties, memory performance bottlenecks, and upper limits for the achievable floating point performance of multigrid methods are determined. The fundamental principles of data locality optimizations based on data access and data layout transformations are introduced in Chapter 4. Although these kinds of optimizations are able to improve the performance of simple codes, they fail to improve the performance of multigrid codes due to data dependences. However, the techniques are the basis of new data locality optimization techniques for the red–black Gauss–Seidel smoother which are proposed in Chapter 5. The performance improvement is demonstrated by means of detailed runtime and memory access analysis. The cache–optimized smoother routines as well as cache–optimized inter–grid transfer operations have been integrated in the DiME-PACK multigrid library which is introduced in Chapter 6. They establish the core routines for the library. In Chapter 7, existing tool support for data locality optimization is discussed and new cache visualization techniques are proposed which are derived from the experience gained during the performance study of multigrid codes. The thesis concludes with some final remarks and a brief outlook on future work.

Chapter 2

Memory Hierarchy Aspects in Computer Architecture

Microprocessors nowadays operate at a clock frequency of one GHz and above. In contrast to the past, however, high performance microprocessors are no longer limited to expensive high end products. Mass market microprocessor products like the Pentium 3 [Int00b, Int00c], Pentium 4 [Int01] and AMD Athlon [Adv99] can be bought in the shop around the corner for reasonable prices and used in common PCs. They provide a floating point peak performance equal or even higher than high end workstations. Hence, new PCs provide a peak performance of one billion floating point instructions per second ($\equiv$ one GFLOPS)and beyond at home for a reasonable price.

The reason for the incredible performance increase is the continuing advance in integrated circuit (IC) technology over the years. 0.18–micron IC process is standard nowadays and microprocessors produced with 0.15–micron and 0.13–micron IC processes already start to appear (see Table 2.1). Along with the continuing miniaturization of circuits the numbers of transistors which can be realized on a chip has been increasing over the years and this trend continues. For example, The HP PA–8500 and HP PA–8600 microprocessors are built with 130 million transistors. Even mass market products like the Pentium 4 and AMD Athlon microprocessor already use a transistor budget of more than 30 million transistors. The average transistor budget of a CPU is still approximately 10 to 20 million transistors (see Table 2.1) but microprocessors with billions of transistors have already been forecasted [BG97]. Microprocessor manufacturers employ these transistors to implement more and more fine–grain parallel instruction execution techniques such as deep pipelining, multiple instruction issue, dynamic scheduling, out–of–order execution, speculative execution, and also on–chip caches with small capacities as compared to main memory. These structural enhancements are additionally contributing to the performance improvement.

In this chapter, the architectural features *pipelining* and *superscalar execution* will be introduced which are among the most important architectural improvements in microprocessor architecture. Then, the limiting performance factor in todays computer architecture — the memory bottleneck — is identified. Subsequently, the memory hierarchy concept

Processor	Clock Rate	Transistors	IC Process	Pipeline
Intel Pentium 4	1.5 GHz	42.0 mio.	0.18μ	22/24 stages
AMD Athlon	1.2 GHz	37.0 mio.	0.18μ	9/11 stages
Intel Pentium 3	1.0 GHz	24.0 mio.	0.18μ	12/14 stages
Sun Ultra–3	900 MHz	29.0 mio.	0.15μ	14/15 stages
Alpha 21264B	833 MHz	15.4 mio.	0.18μ	7/9 stages
HP PA–8600	552 MHz	130.0 mio.	0.25μ	7/9 stages
Sun Ultra–2	480 MHz	3.8 mio.	0.29μ	6/9 stages
IBM Power 3–II	450 MHz	23.0 mio.	0.22μ	12/14 stages
MIPS R12000	400 MHz	7.2 mio.	0.25μ	6 stages

Table 2.1: Some parameters of currently available microprocessor chips in 2001[Mic00].

is described which is used by computer manufacturers to mitigate the impact of the memory bottleneck. Then, an overview of the state of the art in microprocessor architecture with an emphasis on the memory hierarchy architecture is given. The chapter concludes with a summary of trends in microprocessor architecture.

2.1 Pipelining and Superscalar Execution

In this section, pipelining will only be briefly introduced. A more detailed description of pipelining for microprocessors can be found in [Kog81, HP96].

Pipelining is a technique that is applied to many situations to speed up the overall execution of a process which repeatedly performs a certain task. It must be possible to divide the task into a series of individual and independent operations, or stages, that, when applied sequentially, perform the overall task. With pipelining the individual operations are executed in an overlapped manner. As one item progresses through the pipeline, other items can be initiated before the first has completed all stages. That is, once the first stage of the first task is completed, the second stage of the first task is executed concurrently with the first stage of the second task, and so on.

Pipelining exploits parallelism among instructions in a sequential instruction stream to speed up execution. In a microprocessor each pipeline stage executes one part of an instruction like instruction fetch (IF), instruction decoding (ID), execution (EX), memory access (MA), and write back (WB). Figure 2.1 illustrates different execution schemes. If instructions are executed sequentially the processor has to execute each phase of the first instruction before it can start executing the next instruction. In a pipelined CPU the execution of the first phase of the second instruction can begin as soon as the first phase of the first instruction is finished.

The longest running stage in a pipeline determines the time (usually measured in clock ticks) to advance all instructions to the next stage. Hence, the work done in each stage should be distributed equally. If this is the case the execution time of one instruction is

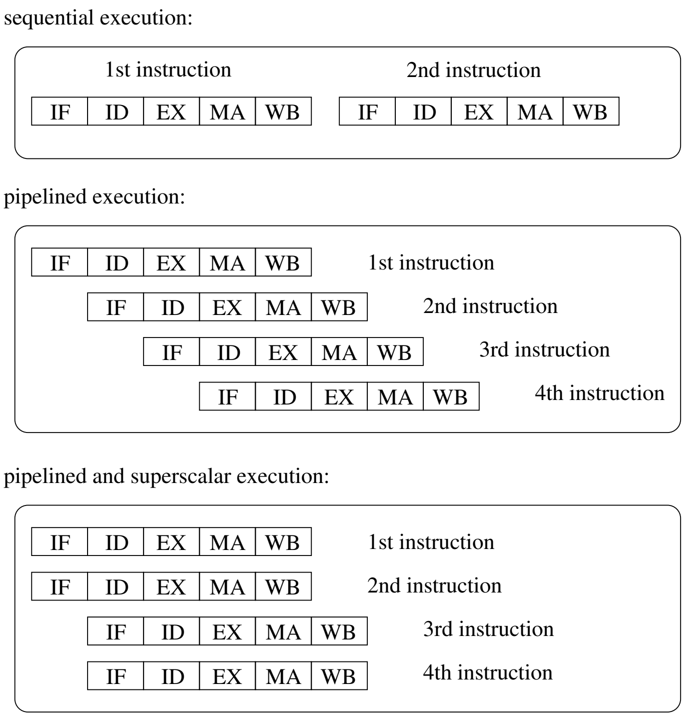

Figure 2.1: Sequential execution of instruction vs. pipelined and superscalar execution

equal to the sequential execution time. In real systems, however, the execution time for one instruction is usually higher than the one for sequential execution. Nevertheless, the throughput of instructions will be increased. Provided that an ideal pipeline is considered, i.e. the work in each stage is equally distributed and no further overhead is present, a pipelined microprocessor will accelerate execution by a factor equal to the number of stages.

The Compaq Alpha 21264 microprocessor [Com99, Gwe96a] implements a 7 stage pipeline for instruction execution, for example. The stages include instruction fetch, instruction slot, mapping, issue, register read, execute, and data cache access. Other processors like the HP PA–8500, the Sun UltraSparc–II, the Power PC G4, or the MIPS R12000 implement pipelines of similar depth. With the increase in clock frequency the time which can be spent in one pipeline stage is decreased. Consequently, some operations which have been completed in one cycle in the past will require several cycles when the CPU is operated at higher frequency. Thus, the stages which process such operations have to be split into several stages leading to an architecture such as the Pentium 4 architecture developed by Intel which already uses a very deep pipeline with 24 stages.

The idea of pipelining only works if there is a continuous flow of instructions through them. If, for whatever, reason a pipeline stage cannot complete its work there will be a long delay until new instructions can be processed. Such a situation is called a *hazard*. Possible hazards are *structural*, *control* and *data hazards*. A more detailed description of hazards can be found in the literature [HP96].

The concept of pipelining can be extended to groups of instructions. Figure 2.1 illustrates the instruction execution of a microprocessor which is able to start the execution of two instructions per cycle. A microprocessor which is able to execute multiple instruction per cycle is called superscalar. Typically, only a small number of instructions — like one to eight — can be executed simultaneously. Parallel instruction execution requires that the instructions are independent of each other. The independence is checked by the microprocessor dynamically during the execution of the instructions.

To be able to process several instructions in parallel the microprocessor must provide the resources needed in each stage like integer and floating point calculation units multiple times. To reduce the number of resources which have to be duplicated, simultaneously executed instructions typically must obey some constraints. A typical constraint, for example, is that only a certain number (usually two to four) of integer instructions can be issued together with a certain number of floating point instructions (usually two).

The Compaq Alpha 21264 microprocessor is able to issue up to four instructions per cycle, for example. It provides two floating point units and four integer execution units (two general–purpose units and two address arithmetic logic units (ALUs)). Hence, although four instructions can be issued simultaneously, only two floating point instructions are allowed per cycle; and also only for certain combinations of integer operations four instructions can be issued per cycle.

2.2 The Bottleneck: Memory Performance

Baskett [Bas91] estimated in 1991 that the performance of microprocessors increased by 80 per cent per year in the past. At the same time the access delays for DRAM chips, however, decreased more slowly at an annual rate of 5 – 10 per cent [ASW+93]. The speed of microprocessors as well as the speed of DRAM chips improved exponentially during the same time. Unfortunately, the exponent for microprocessors is substantially larger than that for DRAM chips. The trend already produced a large gap between CPU and DRAM speed. Although the annual performance of microprocessors slightly decreased in the last years, the general trend is still maintained so that the gap will grow further. The significance of this trend will be illustrated with the following example.

At a clock frequency of one GHz a two–way superscalar CPU is theoretically able to perform two floating point operations every nanosecond. For each floating point operation two words are required as operands and one word is produced as a result. With a memory access latency of about 100 nanoseconds which is approximately the memory latency of the Compaq PWS 500au, for example, a microprocessor will face a memory access latency of approximately 100 cycles every time it fetches data from main memory. This is a severe problem since the microprocessor could execute up to 200 floating point instructions during that time. This problem is called *latency problem*. Researchers have developed several techniques, like software and hardware prefetching [CKP91, MLG92, CB94], non–blocking caches [Kro81, SF91], stream buffers [Jou90, PK94], multithread-

Processor	Bandwidth	Out–of–Order	Cache (I/D/L2)
Sun Ultra–3	4.8 Gbyte/s	none	32 K / 64 K / –
Intel Pentium 4	3.2 Gbyte/s	126 ROPs[1]	12 K / 8 K / 256 K
Alpha 21264B	2.7 Gbyte/s	80 instr	64 K / 64 K / –
AMD Athlon	2.1 Gbyte/s	72 ROPs	64 K / 64 K / 256 K
Sun Ultra–2	1.9 Gbyte/s	none	16 K / 16 K / –
IBM Power 3–II	1.6 Gbyte/s	32 instr	32 K / 64 K / –
HP PA–8600	1.5 Gbyte/s	56 instr	512 K / 1 M / –
Intel Pentium 3	1.1 Gbyte/s	40 ROPs	16 K / 16 K / 256 K
MIPS R12000	0.5 Gbyte/s	48 instr	32 K / 32 K / –

Table 2.2: Memory peak bandwidth, out–of–order capability, and on–chip cache sizes of microprocessor chips in 2001 [Mic00].

ing [LGH94], and out–of–order execution [HP96] to tolerate at least some memory access latency. However, these techniques are not able to compensate a latency of over 100 cycles. Especially, since instructions executed while loading data might access other data themselves which may again lead to idle time which has to be tolerated. Another problem involved with some latency tolerating techniques such as prefetching, for example, is that they increase the total amount of memory traffic and thus will expose the *memory bandwidth problem* [BGK95].

To store the results of two floating point operations per cycle in memory, a microprocessor requires a memory bandwidth of six words each nanosecond, or 48 Gbyte/s (assuming double precision floating point operations). Although the focus of computer designers has recently shifted towards increasing the memory bandwidth, the peak main memory bandwidth of today's computers is still far below 48 Gbyte/s. As Table 2.2 shows the peak bandwidth of microprocessors is typically between one and two Gbyte/s. The peak bandwidth, however, is a theoretical value which is very hard to achieve in real life.

Another measure of bandwidth is the sustainable bandwidth of user programs which is determined with the STREAM benchmark [McC95]. The STREAM benchmark program accesses data in a way which is advantageous for memory systems. Thus, the memory bandwidth achieved with it can be seen as the maximally achievable user program memory bandwidth. Figure 2.2 summarizes the results of the STREAM benchmark on a HP SPP 2200 Convex Exemplar node (HP PA–8200, 200 MHz), a Compaq PWS 500au (Alpha 21164, 500 MHz), a Dell PC (Intel Pentium 2 Xeon, 450 MHz), a SGI Origin 2000 node (SGI R10000, 195 MHz), a Sun Ultra 60 (UltraSPARC–II, 296 MHz), a HP N–Class node (HP PA–8500, 440 MHz), and a Compaq XP1000 (Alpha 21264, 500 MHz). The Compaq XP1000 with a peak bandwidth of over two Gbyte/s reaches only a STREAM bandwidth of 745 Mbyte/s. As Figure 2.2 shows the sustainable bandwidth for other

[1]The Intel Pentium 4, Intel Pentium 3, and AMD Athlon are x86–compatible microprocessors. Thus, they internally translate the complex x86 instructions into possibly several RISC operations (ROPs) which are then executed by the microprocessor core.

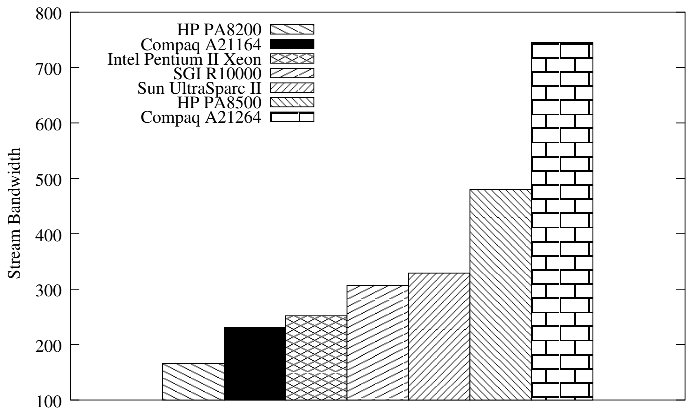

Figure 2.2: Stream Bandwidth for several workstations

architectures might be even less.

Current computers are already slowed down considerably by the slower main memory. If the difference in microprocessor speed and main memory speed diverges at the same pace, a point will be reached where the speed of calculations is determined by the speed of memory. Improving microprocessor speed will then only result in marginal improvement in execution speed. This phenomenon is called *hitting the memory wall* [WM95]. The typical solution is to postpone the impact to a later date by implementing the concept of a *memory hierarchy*.

2.3 The Memory Hierarchy Concept

There are many different ways of implementing the *memory hierarchy* concept, but the consistent theme is that there is a small, expensive, high speed memory at the top of the hierarchy which is usually integrated within the CPU to provide data with low latency and high bandwidth. As we move further away from the CPU the layers of memories get successively larger and slower. These high speed memories are called *caches* and are intended to contain copies of main memory blocks to speed up accesses to frequently needed data. The lowest level of the memory hierarchy is the main memory which is large but also comparatively slow. The levels of the memory hierarchy usually subset one another so that data residing within a smaller memory is also stored within the larger memories.

A typical memory hierarchy is shown in Figure 2.3. It contains a small number (32 to 64) of registers on the chip which are accessible without delay. Furthermore, a small cache — usually called level one (L1) cache — is placed on the chip to ensure low latency and high bandwidth. The L1 cache is usually split into two separated caches. One only keeps data, the other instructions. The latency of on–chip caches is typically one or two cycles. The chip designers, however, already face the problem that large on–chip caches of new high–frequency microprocessors aren't able to deliver data within one cycle since

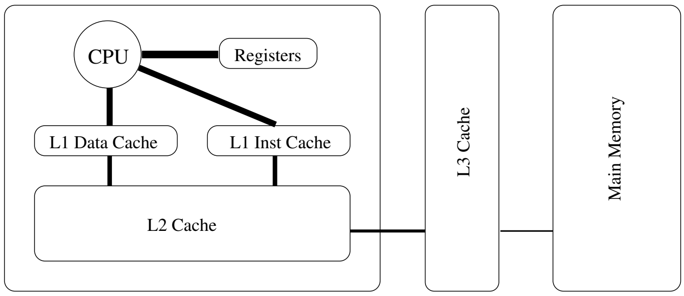

Figure 2.3: A typical memory hierarchy containing two level one on–chip caches, one on–chip level two caches, and a third level of cache off–chip. The thickness of the interconnections emblematizes the bandwidth between levels of the memory hierarchy.

the signal delays are too long. Therefore, the size of on–chip L1 caches is limited to 64 Kbyte or even less for many chip designs. Larger cache sizes with accordingly higher access latency, however, start to appear. Table 2.2 summarizes on–chip cache sizes of some state of the art microprocessors.

The L1 caches are usually backed up by a level two (L2) cache. In some architectures like for example the Compaq Alpha 21164 [Com97], the Intel Pentium 4 [Int01] or the AMD Athlon [Adv99] the second level cache is also implemented on chip. Most of the architectures still build the L2 cache with SRAM chips on the motherboard of the computer. Off–chip caches are much bigger but also provide data with lower bandwidth and higher access latency. L2 caches on chip are usually smaller than 512 Kbyte and provide data with a latency of approximately 5 to 10 cycles. If the L2 caches are implemented on the chip, a third level of off–chip cache may be added to the hierarchy. Off–chip cache sizes vary from one Mbyte to 16 Mbyte. They tend to provide data with a latency of 10 to 20 cycles.

2.3.1 Locality of References

Because of the limited size, caches can only hold copies of recently used data or code. Typically, when new data is loaded into the cache other data has to be replaced[2]. Caches improve performance only if data already loaded into the cache is *reused* before it is replaced by other data. The reason why caches are nevertheless able to substantially reduce program execution time is the *principle of locality of references* [HP96] which states that recently used data is very likely to be used again in the near future. Locality can be subdivided into *temporal locality* and *spatial locality* [WL91]. A sequence of references has temporal locality when recently accessed data is likely to be accessed again

[2]In the startup phase of a microprocessor or after a cache flush no data has to be replaced since the cache is considered to be empty. Some set–associative caches may also voluntarily evict data, so that empty cache lines are present in the cache. To simplify matters, the further explanation ignores these special cases. A description of the handling of these cases can be found in [HP96], for example.

in the near future. A sequence of reference has spatial locality when data located close together also tends to be referenced close together in time.

2.3.2 Block Placement

Data within the cache is stored in *cache lines*. A cache line holds the content of a contiguous block of main memory. If data requested by the processor is found within a cache line this is called a *cache hit*. Otherwise, a *cache miss* occurs. The content of a *memory block* containing the requested word is fetched from a lower memory layer and copied into a cache line. In that process another item is (typically) replaced. Therefore, the question into which cache line the data should be placed and how to locate it must be handled efficiently to guarantee a low latency.

One of the cheapest approaches to implement block placement strategies in respect to hardware expense is *direct–mapping*. Thereby, the contents of a memory block can be placed (or mapped) into exactly one cache line. The cache line in which the contents will be copied is determined by the following formula:

cache line address = (block address) MOD (number of cache lines)

Direct–mapped caches have been among the most popular cache architectures in the past. Recently, however, computer architects returned to implement more and more k–way set–associative on–chip caches. Direct–mapped caches, however, are still very common for off–chip caches. k–way set–associative caches have higher hardware expenses but usually imply higher hit rates.

The cache lines of *k–way set–associative* caches are grouped into sets of k cache lines and the contents of a memory block can be placed into any cache line of exactly one set. The set in which the contents will be placed is determined by the following equation:

set address = (block address) MOD (number of sets)

Finally, a cache is *fully associative* when the contents of a memory block can be placed into any cache line of the cache. Fully associative caches are usually only implemented in small sized special purpose caches, but not in general data or instruction caches.

Direct–mapped and fully associative caches can be viewed as special cases of k–way set–associative caches. A direct–mapped cache is a one–way set–associative cache and a fully associative cache is an n–way set–associative cache provided that n equals the number of cache lines.

2.3.3 Block Replacement

In fully associative and k–way set–associative caches the memory block can be placed in several cache lines. The question into which cache line a new memory block is copied and consequently which block has to be replaced is decided by a *(block) replacement strategy*. The two most commonly used strategies for today's microprocessor caches are *random* and *least–recently used* (LRU). The random replacement strategy chooses a random cache line to be replaced. The LRU strategy replaces the block which hasn't been accessed for the longest time. According to the principle of locality it is more likely that a recently accessed item will be accessed again in the near future. Less common strategies are the *least–frequently used* (LFU) and *first in, first out* (FIFO) replacement strategy. The former replaces the item which was least frequently used whereas the latter replaces the memory block in the cache line which resided in the cache for the longest time.

Finally, the *optimal* replacement strategy [Bel66, SA93] replaces the memory block which will not be accessed for the longest time. It is impossible to implement this strategy in a real cache, since it requires information about future cache references. Thus, the strategy is only of theoretical value. Note, however, that a fully associative cache with optimal replacement strategy will have the minimal number of cache misses any cache of the same size for any possible sequence of references can have [SA93, Tem98].

2.4 State of the Art

In the following, an overview of the state of the art in microprocessor technology is given with an emphasis on the cache and memory system. In addition, predecessor models and some already announced, but not yet available, microprocessors are introduced briefly. Readers not interested in technical details may skip the following sections and resume the reading in Section 2.4.7.

2.4.1 Compaq Microprocessors

The microprocessor currently shipped by Compaq is the Alpha 21264 [Gwe96a]. It is the successor to the Alpha 21164 [Com97]. For the Alpha 21164 the chip manufacturer decided to use a very simple design without out–of–order execution, branch prediction, or other sophisticated features which were state of the art at the time the Alpha 21164 was released. The simple design was chosen to allow the chip to be operated at a clock frequency that was much higher than that of other microprocessor at that time.

Alpha 21164

The Alpha 21164 is equipped with a direct–mapped 8 Kbyte instruction and a direct–mapped 8 Kbyte data cache, both located on the chip. Furthermore, a unified three–way set–associative 96 Kbyte second level cache for data and instruction is implemented on chip. The design allows a L1 data access with only one cycle latency. A second level cache access already takes 6 to 10 cycles. Finally, the L2 cache is backed up by an optional direct–mapped off–chip cache of up to 64 Mbyte. Data from main memory can be delivered to the microprocessor resp. L3 cache if present with a peak memory bandwidth of 400 Mbyte/sec.

Alpha 21264

In contrast to its predecessor, the Alpha 21264 implements state of the art processor features. The Alpha 21264 implements out–of–order execution of 80 instructions and allows 8 outstanding loads. It also provides branch prediction techniques to reduce idle time. The execution core is equipped with four integer and two floating point arithmetic units. It is able to execute up to six instructions per cycle. However, only four instructions can be issued per cycle so that only an execution of four instructions can be sustained. The drawback of the new design is that it only allows a marginal increase of the clock frequency and the peak performance of the chip is equal to the peak performance of its predecessor, operated with the same clock frequency. Nevertheless, the sustained performance for many applications increased substantially.

One reason for this is, that the designers spent a lot of effort into improving the memory access performance of the new chip. The Alpha 21264 is equipped with two separated 64 Kbyte on–chip L1 caches for instructions and data. Both caches are two–way set–associative. The instruction cache uses 32 byte cache lines whereas the data cache uses 64 byte cache lines. The designers decided to build two large L1 caches for instruction and data instead of two small caches backed up by an additional second level of on–chip cache. The access latency of the primary data caches, however, is two to four cycles and not one cycle like for the L1 cache of the Alpha 21164 microprocessor. The data cache is dual ported and can deliver two independent 64–bit words every cycle. The microprocessor includes a dedicated 128–bit L2 cache bus and a separated 64–bit system bus to main memory. Both can be operated at speeds up to 333 MHz. The external cache provides a backup store for both primary caches. It is direct–mapped and shared by instructions and data. The off–chip cache is controlled by a cache controller on the processor to guarantee a low latency. The off–chip cache size can vary from one to 16 Mbyte and the latency is 12 to 14 cycles. Data from the second level off–chip cache can be delivered to the primary data caches with a peak bandwidth of 4 Gbyte/s with 250 MHz SRAM chips. For the Alpha CPU also a slot architecture with processor and L2 cache on a small PC board similar to Intel's Slot 2 Xeon is available. This allows the off–chip cache to be operated at a higher clock frequency to reduce latency and increase bandwidth. The system bus which connects the chip with main memory is operated at 333 MHz and delivers a peak main memory bandwidth of 2.7 Gbyte/s.

Alpha 21364

The successor to the Alpha 21264, the Alpha 21364, is announced for the second half of 2001 [Gwe98a]. It will be based on a the Alpha 21264 CPU with additional features wrapped around the core. The main improvement will be the use of multiple Direct RDRAM channels which will increase main memory bandwidth and greatly reduce memory access latency. Furthermore, a six–way set–associative 1.5 Mbyte L2 cache will be placed on chip. The cache will cycle at the speed of the CPU, delivering 128 byte of data every nanosecond. The latency of the L2 cache will be the same as the second level cache latency of the Alpha 21264. This is a direct consequence of the decision that the 21364 uses the same core design with only minor changes. In contrast to the other Compaq chips the Alpha 21364 will not support any off–chip caches. However, the RDRAM channels will provide a direct main memory access with a peak bandwidth of 6.0 Gbyte/s

2.4.2 Hewlett Packard Microprocessors

The current microprocessor shipped by Hewlett–Packard is the HP PA–8600 [Gwe99c]. It is a member of the PA–8x00 family which is based on the core design of the PA–8000 microprocessor.

PA–8000

The PA–8000 [Hun95] is a four–way superscalar microprocessor with dynamic instruction reordering capability (out–of–order execution). The PA–8000 includes two integer ALUs and two floating point multiply/accumulate units. Hence, the chip is able to execute up to four floating point operations per cycle in the case of multiply–add operations. Otherwise two floating point operations and two integer operations can be performed in parallel.

The memory hierarchy design of the PA–8x00 family is very conservative. It is limited to a single level of two large low latency caches for data and instructions. The PA–8000, for example, is equipped with a one Mbyte two cycle latency direct–mapped off–chip cache for both data and instructions.

PA–8200

The PA–8200 microprocessor [SBK+97, Gwe96b] uses a fine tuned version of the PA–8000 core. The cache design is more or less equal to the design of the PA–8000, however, newer SRAM technology allows a cache size of two Mbyte for both data and instructions.

PA–8500

The PA–8500 processor [LH97] is based on the PA–8200 core with only minor enhancements. The use of a newer fabrication process allows a larger amount of transistors for the PA–8500 microprocessor. The enhanced transistor budget is used to move the two first–level caches on chip. Therefore, the PA–8500 includes a 0.5 Mbyte four–way set–associative cache for instructions and a one Mbyte four–way set–associative cache for data. Further off–chip caches are not supported. The latency of both caches is still two cycles, but the new design allows a higher clock frequency for the cache and consequently a higher clock frequency for the CPU core. The drawback of the decision to move both caches on chip, however, is a dramatically increased die size and consequently higher manufacturing costs. Consequently, the PA–8500 microprocessor as well as its successor the PA–8600 microprocessor currently have the largest die size of all microprocessors available.

PA–8600

The PA–8600 microprocessor [Gwe99c] in turn is based on the PA–8500 core. Improvements in the fabrication process allow slightly higher clock frequencies. The CPU is equipped with two on–chip caches of the same size as the PA–8500. The cache system is also more or less unchanged. However, the cache prefetching algorithm was improved and the cache replacement was changed from round–robin to LRU replacement.

PA–8700

The next microprocessor — the PA–8700 [Gwe98b, Kre00a] — from Hewlett Packard is expected to be shipped in 2001. The design will be based on the PA–8600 design with only minor modifications to integrate the new cache sizes of 1.5 Mbyte of L1 on–chip data cache and 0.75 Mbyte on–chip instruction cache. Furthermore, again minor improvements will be made in the data prefetching algorithm and the replacement strategy will be changed to quasi LRU replacement. After 2002, the PA–8x00 microprocessor family will be replaced by IA–64 based CPUs.

2.4.3 Sun Microsystems Microprocessors

UltraSparc–II

The UltraSparc–II [Sun97] is the predecessor of the UltraSparc–III chip which is currently shipped by Sun Microsystems. It is equipped with two separate 16 Kbyte caches on the chip for instructions and data. The instruction cache is pseudo associative and the line size is 32 bytes[3]. The data cache is direct mapped and the cache line size is also 32 bytes. Both caches are backed up by a unified direct–mapped external L2 cache of up to 16 Mbyte with a cache line size of 64 byte. The external cache is controlled by an on–chip integrated cache controller.

UltraSparc–III

The UltraSparc–III [Son97b, Kre00c, HL99] chip is based on a completely new core design. The new core supports four–way superscalar in–order execution. The core operates a pipeline with 14 stages to allow a high clock frequency. The chip is equipped with a 32 Kbyte four–way set–associative instruction cache using pseudo random replacement. The instruction cache uses a cache line size of 32 bytes and provides data with a two cycles latency. Furthermore, the chip is equipped with three first level on–chip data caches: a 64 Kbyte four–way set–associative general purpose data cache using pseudo random replacement and 32 byte cache lines, a two Kbyte four–way set–associative write cache

[3]Upon a memory request a pseudo associative cache looks up a memory block in cache like a direct mapped cache. If the data is not present, i.e. a cache miss happens, a secondary slot is checked for the data. If it is found there the cache lines are swapped. Thus, a pseudo associative cache behaves like a two–way set–associative cache but has a shorter access time.

using LRU replacement and 64 byte cache lines to reduce store traffic to the external L2 cache, and a two Kbyte four–way set–associative prefetch cache using LRU replacement. The L1 general purpose data cache provides data with a two cycle latency. If data is not found in the L1 general purpose data cache but in the prefetch cache, the prefetch cache is able to provide the data for two load instructions per cycle with a latency of three cycles. Furthermore, the prefetch cache fills its 64–byte cache lines in 14 cycles from the L2 cache. The prefetch cache is able to process up to 8 software– and hardware–initiated prefetch requests simultaneously.

The on–chip caches are backed up by an external direct–mapped L2 cache which can be up to 8 Mbyte large. The cache is managed by a cache controller which is on the chip to keep latency low [HL99]. The peak bandwidth of the L2 cache is 6.4 Gbyte/sec with 200 MHz SRAM chips. To reduce main memory latency the microprocessor also includes an SDRAM controller which delivers a main memory peak bandwidth of 2.4 Gbyte/sec.

MAJC-5200

Besides the SPARC–architecture Sun Microsystems also developed the MAJC (Microprocessor Architecture for Java Computing) architecture [Sun99]. The architecture is a very long instruction word (VLIW) architecture [Kar93] which allows several instructions to be grouped into one package which is then issued to functional units. Note, that instructions are grouped into packages at compile time to be executed concurrently at runtime. Contrary, superscalar RISC architectures use complex control logic in order to decide dynamically at run time which operation can be executed concurrently. To provide a Java–friendly environment MAJC also supports thread–level parallelism by providing several CPU cores on a chip. Instructions of different threads are bundled into different packages which can then be dispatched to different CPU cores.

The MAJC–5200 [Sud00, Cas99] is the first (and so far only) implementation of MAJC. It implements two identical and independent but cooperative processor cores. Each core is equipped with a 16 Kbyte, two–way set–associative (LRU replacement) instruction cache using 32 byte lines. Furthermore, both cores share a single 16 Kbyte, four–way set–associative (LRU replacement) data cache with 32 byte cache lines. The data cache is dual ported to allow the simultaneous access of both cores. The design does not include a L2 cache. In order to compensate for the missing L2 cache the chip is equipped with four IO–ports. Two of the ports are general purpose off–chip communication paths with a bandwidth of 2 Gbyte/sec each. Furthermore, the chip is equipped with a Rambus DRAM (RDRAM) interface which is able to deliver data from main memory with a bandwidth of up to 1.6 Gbyte/sec. Finally, the chip is equipped with a PCI interface. The high off-chip bandwidth is supposed to give the chip an advantage in multimedia processing where typically large amounts of streaming data have to be processed.

2.4.4 IBM Microprocessors

Power 3

The Power 3 [Son97a, PDM$^+$98] processor from IBM implements a two level memory hierarchy. The first level instruction and first level data cache are located on chip, whereas the unified L2 cache is located off chip. The L1 instruction cache is 32 Kbyte large, using 128 byte cache lines. The latency of the instruction cache is one cycle. The L1 data cache is 64 Kbyte large and uses 128 byte lines. According to [PDM$^+$98] both L1 caches are 128–way set–associative. The instruction cache allows two outstanding loads, whereas the data cache allows four outstanding loads. The controller for the off–chip L2 cache is placed on board. The L2 cache controller allows a maximum of 16 Mbyte of cache. At 200 MHz the bandwidth from the L2 cache to the processor is 6.4 Gbyte/sec. The chip also implements a hardware initiated prefetching. Sequential instruction or data accesses are detected in hardware. A stream is detected when the data accesses happen with a stride of one cache line. The Power 3 allows four streams to be prefetched with up to two cache lines fetched ahead from L2 or main memory.

Power 4

The successor to the Power 3 microprocessor will be the Power 4 microprocessor [Die99b, Kre00b]. It will supports multithreaded programs by implementing chip multiprocessing (CMP) and provides a high bandwidth chip–to–chip interconnection network. The microprocessor chip will include two identical processor cores on chip. Each will be equipped with a 64 Kbyte L1 instruction cache and 32 Kbyte L1 data cache. The core will allow 11 outstanding loads (eight from the data cache and three from instruction cache). The caches of both cores will be backed up by a unified L2 cache which will be split into three equally sized, and independent 0.5 Mbyte caches. Two slices will be dedicated for the two cores. The remaining slice will be dedicated for the chip–to–chip interconnect. The slices will be connected to the cores by a 100 Gbyte/sec port to allow high bandwidth from caches. The L2 slices will be ensured to be cache coherent. All L2 slices together will allow a total of 12 outstanding loads from L3 or main memory. Furthermore, the chip will include a cache controller for an eight–way set–associative off–chip cache. The controller will deliver data from L3 cache with a bandwidth of 10 Mbyte/sec. Similar to the Power 3 microprocessor, the Power 4 chip will also implement hardware prefetching. The Power 4 chip will allow eight prefetch streams with up to 20 cache lines kept in flight. Finally, the chip will be equipped with a chip–to–chip interconnect which allows high–speed coupling with two other Power 4 microprocessors. The chip–to–chip interconnect will allow a sustained data transfer bandwidth of 35 Gbyte/sec from chip to chip.

2.4.5 Intel Microprocessors

Pentium 3

The Pentium 3 processor [Gwe99a, Int00c, Int00b] implements a two level cache hierarchy. The first level of the hierarchy includes a 16 Kbyte, four–way set–associative L1 instruction cache and a 16 Kbyte, four–way set–associative L1 data cache on chip. The L2 cache contains both data and instructions. The L2 cache is available in two different variants [Gwe99a]: a 512 Kbyte, four–way set–associative off–chip cache or a 256 Kbyte, eight–way set–associative, on–chip cache.

Pentium 4

The successor chip — the Pentium 4 [Sti00, Gla00] — also implements a two level cache hierarchy. The first level of the hierarchy includes a 8 Kbyte four–way set–associative data cache using 64 byte lines and a newly designed instruction cache called *trace cache*. The Pentium microprocessor family decodes x86 instructions into several micro operations which can then be executed internally in a RISC style manner. The trace cache does not store x86 instructions like the older Pentium instruction cache designs do, but already decoded micro operations. The trace cache is able to cache 12K of micro operations. The second level cache is a unified eight–way set–associative L2 cache of 256 Kbyte. The L2 cache uses 128 byte cache lines and is integrated on chip. An option for an L3 off–chip cache is planned to appear in the next version of the core. The chip is connected to main memory with a 400 MHz bus. Instead of DRAM or SDRAM, so far only RDRAM with a peak bandwidth of 3.2 Gbyte/sec is supported.

Itanium

Besides the Pentium architectures Intel also developed the IA-64 based on EPIC. EPIC (Explicitly Parallel Instruction Computing) is a 64-bit microprocessor instruction set, jointly defined and designed by Hewlett Packard and Intel, that provides up to 128 general and floating point unit registers and uses speculative loading, predication, and explicit parallelism (VLIW style). The Itanium processor [Int00a] is the first implementation of the IA-64 architecture based on EPIC. In contrast to the Pentium systems the Itanium processor is not equipped with a out–of–order execution RISC core but executes instructions in software–supplied order.

The Itanium implements a three level cache hierarchy. The first two cache levels are integrated on the chip whereas the third level is located off–chip. The first level of cache consists of a 16 Kbyte instruction cache and a 16 Kbyte data cache. Both first level caches are four–way set–associative and use 32 byte cache lines. The L1 data cache is only used for integer data. Thus, floating point load instructions bypass the L1 data cache and fetch data directly from the L2 cache. The L2 cache is 96 Kbyte large, 6–way set-associative, and uses 64 byte cache lines. The L2 caches is unified, i.e. it caches data as well as instructions. The L3 cache is located off-chip but integrated in the Itanium cartridge.

Depending on the particular Itanium package two Mbyte or four Mbyte L3 cache are supported, respectively. The latency for integer loads is two cycles for a L1, six cycles for a L2, and 21 cycles for a L3 cache access. The latency for floating point data loads is slightly higher: 9 cycles for a L2 and 24 cycles for a L3 cache access.

2.4.6 AMD Microprocessors

AMD currently sells AMD Athlon and AMD Duron microprocessors. The AMD Athlon microprocessor is available with two different cores: the K75 [Adv99] and the Thunderbird [Adv00b] core.

Athlon K75 core

The K75 microprocessor core implements a two level cache hierarchy. The first level of the memory hierarchy includes a 64 Kbyte two–way set–associative L1 instruction cache and a two–way set–associative L2 data cache. Both caches are backed up by a direct–mapped off–chip L2 cache. The L2 cache can be up to 8 Mbyte large. However, the on–chip controller is optimized for a 512 Kbyte L2 cache.

Athlon Thunderbird and Duron

The Thunderbird core also implements a two level cache hierarchy [Adv00a]. However, the Thunderbird core includes both levels on chip. The L1 instruction cache and data cache are both 64 Kbyte large and two–way set–associative. The 256 Kbyte large 16–way set–associative L2 cache delivers data to both first level caches. In contrast to other microprocessor developers, AMD did not choose the L2 cache to be an inclusive cache but an exclusive cache. That is, the L2 cache does not include the data which is present in one of the L1 caches. The AMD Duron microprocessor is identical with the AMD Athlon (Thunderbird core) with the only exception that the L2 cache size is only 64 Kbyte.

2.4.7 Summary of the State of the Art

More or less all microprocessors nowadays are equipped with at least one level of cache. The first level of cache which is usually located on the processor die includes two caches: one cache for data and one for instructions. The size of first level caches varies from 8 Kbyte to one Mbyte. Some microprocessors like the Pentium 4 or the Alpha 21164 have only 8 Kbyte of cache for data and another 8 Kbyte for instructions. The reason for such small level one caches is that at a clock frequency of 500 MHz to 1.5 GHz the signal delays do no longer allow a one cycle access latency for larger caches. Several microprocessor manufacturers nevertheless build CPU with larger L1 caches even if that implies that the cache will have an access latency of at least two cycles. Examples for CPUs with larger L1 caches are the Compaq Alpha 21264 with 64 Kbyte data and 64 Kbyte instruction cache or the IBM Power 3 chip with 64 Kbyte for data and 32 Kbyte of cache for instructions. HP decided to equip its new microprocessors with a single level of two very large on–chip caches. The HP PA–8500 and the HP PA–8600, for example, include 0.5 Mbyte of instruction cache and one Mbyte of data cache on the processor die. HP also decided that the large L1 caches are sufficient and that no other cache levels are required. Most other microprocessor manufacturers, however, implement one or two additional levels of cache. Intel equips the Pentium 4 with another second level of cache on the chip with 256 Kbyte for data and instructions, for example. However, the chip does not support any off–chip cache. The Compaq Alpha 21164, on the other hand, is equipped with a medium–sized second level cache on chip and another level of SRAM cache off chip. Placing cache on the processor die allows the cache to be operated at the same frequency as the processor core. This reduces the access latency to the cache and the tight integration usually allows a higher bandwidth. The drawback is that the microprocessor chip requires more transistors and will be more expensive in manufacturing. However, this might not be a severe obstacle in the future, as the amount of transistors is increasing thanks to improved semiconductor technology.

Contrary, off–chip caches can be built with comparatively cheap SRAM chips. This allows much larger cache sizes but also implies higher latency and lower bandwidth. Therefore, most of the chip designers include on chip controllers for off–chip caches and main memory.

In the recent past most microprocessors used direct–mapped caches. Most of the on–chip caches nowadays, however, are highly associative and even off–chip caches start to be associative due to on–chip cache controllers. Furthermore, cache systems are getting smarter due to automatic stream detection, prefetching, better replacement strategies, and many other features. Sun's UltraSparc–III and IBM's Power 3, for example, support hardware and software initiated prefetch to hide main memory latency.

2.5 Future Processor Trends

Within the next 15 years microprocessors with a billion logic transistors are forecasted [BG97, ITR00]. These microprocessor chips will be clocked with a frequency of several GHz.

The increase in frequency in the past already led to very deep pipelines. The average pipeline currently is 12 to 14 stages deep and even longer pipelines are already implemented in the new high frequency microprocessors like the Pentium 4. The further increase in clock frequency will most likely imply even deeper pipelines. Deep pipelines, however, involve inefficiency once pipeline stalls occur, the workload of the stages cannot be balanced, or if instructions are not inserted into the pipeline every cycle (pipeline bubbles).

In the past further performance gain was achieved through exploitation of instruction level parallelism (ILP).This introduced four–way, out–of–order, speculatively executing CPU cores which already use more than 10 million transistors. This is approximately 10 times as much as a one–way issue core would require. The sustainable ILP, however, typically is two instructions per cycle. That is, the sustainable throughput is typically only two instructions per cycle. Thus, the transistor budget is used inefficiently in respect to performance gain and it will be hard to do better.

Nevertheless, some microprocessor developers claim that with more transistors and sophisticated compiler support it will be possible to exploit more ILP. Future microprocessor architectures which try to exploit more ILP are for example *advanced superscalar processors* [PPE$^+$97], *superspeculative processors* [LS97], and *trace processing* [SV97]. Intel's and HP's EPIC architecture is an example of an advanced superscalar architecture although their approach is not based on a RISC but on a VLIW approach with some features of a superspeculative architecture. The Hal Sparc64 V [Die99a] chip, on the other hand, is based an a trace processing architecture with superspeculative execution.

Other microprocessor manufacturers claim that ILP is already tapped out so that only thread level parallelism (TLP) remains to be exploited. IBM with the Power 4 microprocessor and Sun with the MAJC–5200 will implement a *chip multiprocessing* (CMP) [NO97] architecture with a small number of independent microprocessors on a single chip which communicate through a coherent (on–chip) cache.

Another approach pursued by Compaq is the exploitation of TLP with a simultaneous multithreaded processor (SMT) [EEL$^+$97]. The approach is based on a superscalar core which shared by several threads. Once one thread is delayed due to a stall, another thread is executed on the core to fill idle cycles. The Alpha 21464 [Gwe99b], for example, will use an eight–way issue core which allows several threads to share the common core[4].

A completely different approach is taken by the IRAM processors [KPP$^+$97]. An IRAM processor will couple the traditional vector processor concept with large DRAM banks (rather than SRAM) on the chip to allow high memory bandwidth. Although, the major area of application will be multimedia applications with streaming data the authors also hope that other areas will also benefit from the higher (on–chip) main memory bandwidth.

In summary, the common property of all architectures is that they will have large on–chip memory capacities to provide data with a reasonably low latency and high bandwidth. Two levels of cache on the chip and multi–megabyte level two caches will be the norm. Unfortunately, even the forecasted cache sizes are too small to cache the whole data set used by multigrid methods for realistic problem sizes and resolutions currently in use. Furthermore, it is very likely that the problem sizes will grow further along with the expected performance improvement. Thus, using existing caches efficiently is already crucial and will be even more crucial if the current trends in performance improvement of microprocessors and memory chips prevail.

[4]Note that, with the current market situation (September, 04th 2001) it is not clear whether the Alpha 21464 will actually be released.

2.6 Summary

The increasing gap in microprocessor and main memory speed leads to a latency and bandwidth problem. Microprocessor manufacturers build computer systems which are equipped with memory hierarchies to mitigate the effect of these problems. The study of microprocessors points out that all memory hierarchies are of similar structure. A typical memory hierarchy nowadays has several layers of cache. The first and maybe a second layer of cache is integrated on the chip. Furthermore, currently available computer systems are equipped with larger off–chip caches. However, the size of these caches is far too small to contain data structures used in multigrid codes. Most of the forecasted systems will be equipped with memory structures similar to the memory hierarchies currently in use. Admittedly, those caches will be bigger and smarter. However, the data structures used in multigrid codes are already too big to even fit in the bigger cache sizes of forecasted computer architectures.

Based on the observation that the microprocessor architectures with memory hierarchies are similar, it can be assumed that the data locality optimization techniques which will be proposed in this thesis are of relevance for virtually all microprocessor architectures which employ the memory hierarchy concept, including systems currently in use as well as forecasted systems.

Chapter 3

Memory Characteristics of Multigrid Methods

In this chapter, the basic concept of multigrid methods and their mathematical principles are briefly described. Then, the performance of multigrid methods will be examined. Detailed runtime as well as theoretical studies of the performance of these methods demonstrate the interaction between multigrid algorithms and deep memory hierarchies. Thereby, delays due to main memory accesses will be revealed as the performance bottleneck of multigrid methods and their components on cache based architectures. Furthermore, upper limits for the achievable performance of multigrid methods on RISC based microprocessors are determined by means of theoretical models.

3.1 Introduction to Multigrid Methods

Multigrid methods play an important role in the numerical solution of physical processes. Chemical, technical or physical processes are often mathematically modeled with elliptic partial differential equations (PDEs). Multigrid methods are among the most efficient algorithms for the numerical solution of that type of equation. This chapter will only provide a brief introduction to the essential ideas of the multigrid concept. A more detailed description of multigrid methods and the mathematical background can be found in [Bra84, Hac85, BHM00].

3.1.1 Model Problem and Discretization

This work focuses on boundary value problems of second–order elliptic partial differential equations. The general form of these PDEs in a two–dimensional domain Ω is:

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + Du_x + Eu_y + Fu + G = 0 \tag{3.1}$$

$$AC - B^2 > 0, \forall (x, y) \in \Omega$$

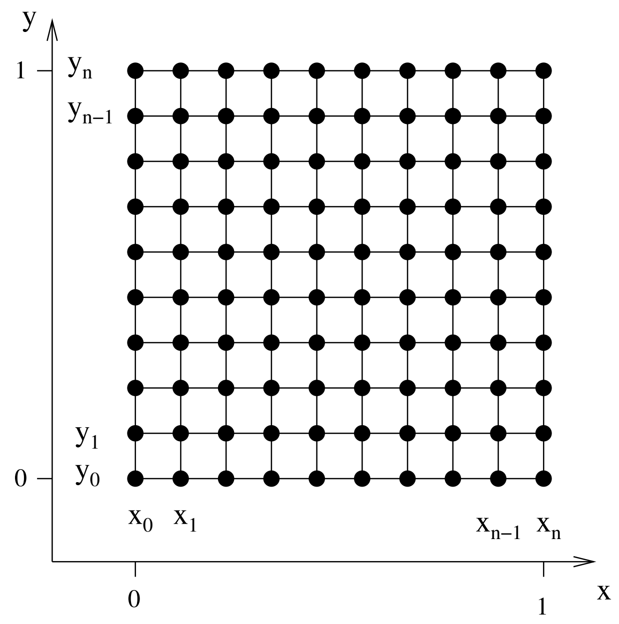

Figure 3.1: Discretization of the domain $(0,1) \times (0,1)$

A,B,C,D,E,F, and G are functions in two real parameters. As an example for illustration the following equation will be used:

$$-\Delta u = -u_{xx} - u_{yy} + \sigma u = f(x,y) \tag{3.2}$$
$$0 < x < 1, 0 < y < 1, \sigma \geq 0$$

With $\sigma = 0$, this equation is usually called *Poisson equation*. The Poisson equation is of relevance for statics calculation, analysis of electrical fields, steady–state temperature distribution in homogenous media, and diffusion processes. In the case of $\sigma > 0$ the equation is called *Helmholtz equation*. For the rest of this chapter, it will be assumed that $\sigma = 0$. The equation is considered under the condition that u is prescribed on the boundary of the domain. This condition is usually called *Dirichlet* boundary condition. Other boundary conditions like *Neumann* or *Cauchy* boundary conditions are not described here.

The numerical solving of a model or real–world problem usually starts with a discretization of the physical continuum. There are different possibilities to discretize the problem domain like for example finite differences, finite elements, or finite volumes. The simplest approach is probably the finite difference method.

For the discretization of the model problem each dimension of the physical continuum ($\Omega = (0,1) \times (0,1)$ in the example) is partitioned into equally spaced subintervals by introducing grid points $(x_i, y_j) = (ih, jh)$ where $h = 1/n$, $i, j = 0, 1, \ldots, n$ with n representing the number of subintervals in each dimension.

The result for the example is a two–dimensional grid over the unit square as shown in Figure 3.1. At each inner point of the grid the differential equation is approximated by a second–order difference equation. The resulting linear equation is:

$$\begin{pmatrix} A & -I & & & \\ -I & A & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & A & -I \\ & & & -I & A \end{pmatrix} \begin{pmatrix} \vec{u}_1 \\ \vec{u}_2 \\ \vdots \\ \vec{u}_{n-2} \\ \vec{u}_{n-1} \end{pmatrix} = \begin{pmatrix} \vec{f}_1 \\ \vec{f}_2 \\ \vdots \\ \vec{f}_{n-2} \\ \vec{f}_{n-1} \end{pmatrix} \tag{3.3}$$

$$I = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & \ddots & & \\ & & & 1 & \\ & & & & 1 \end{pmatrix} A = \begin{pmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{pmatrix}$$

$$\vec{u}_i = (u_{i_1}, u_{i_2}, \ldots, u_{i_{n-2}}, u_{i_{n-1}}), 1 \leq i \leq n-1$$

$$\vec{f}_i = (f_{i_1}, f_{i_2}, \ldots, f_{i_{n-2}}, u_{i_{n-1}}), 1 \leq i \leq n-1$$

The coefficient matrix is sparsely populated and is usually stored in a stencil notation which only represents the relation to neighboring nodes in the grid to reduce storage requirement. The stencil notation for the Poisson equation is:

$$\frac{1}{h^2} \begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix} \tag{3.4}$$

The system of linear equations can be solved with standard elimination methods like Gauss elimination, for example. Unfortunately, Gauss elimination destroys the sparsity of the coefficient matrix by generating fill–ins. The much higher memory requirement of that method, therefore, usually forbids the usage of these methods for larger systems of equations. Furthermore, the complexity of Gauss elimination is relatively high with $O((n-1)^{2\cdot 3}) = O(n^6)$. Better complexity is achieved with band matrix solvers ($O(n^4)$), direct solvers based on Fourier transformation ($O(n^2 \log n)$), or iterative methods.

3.1.2 Basic Iterative Methods

Multigrid methods provide a space and time efficient way to solve systems of linear equations which arise from the numerical solution of PDEs. A conventional iterative method or relaxation method is one of the main components of a multigrid method. Iterative methods generate a sequence of approximations of the exact solution u of a system of equations $Au = f$ (3.5) starting with an initial guess $v^{(0)} = (v_1^{(0)}, v_2^{(0)}, \cdots, v_{m-1}^{(0)})^T$. In this thesis, only the *Jacobi* and *Gauss–Seidel* method together with slightly modified versions of these methods will be introduced. Further iterative methods can be found in the literature [GL83, Hac93].

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m-1} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m-1} \\ \vdots & \vdots & \cdots & \vdots \\ a_{m-1,1} & a_{m-1,2} & \cdots & a_{m-1,m-1} \end{pmatrix} u = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{m-1} \end{pmatrix} f = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{m-1} \end{pmatrix} \quad (3.5)$$

An initial guess can be for example $v^{(0)} = (0, 0, \cdots, 0)^T$. From the initial guess $v^{(0)}$ the sequence of approximations should converge as fast as possible to the exact solution u. The faster the sequence converges the better the iterative method is.

Jacobi Method

Among the simplest iterative methods is the *Jacobi* method. It solves a linear system of equations $Au = f$ by generating a new approximation $v^{(k+1)}$ from the previous approximation $v^{(k)}$ in the following way:

$$v_i^{(k+1)} := \frac{1}{a_{i,i}} \left(- \sum_{j=1, j \neq i}^{m-1} a_{i,j} v_j^{(k)} + f_i \right) \quad (3.6)$$

$$i = 1, 2, \ldots, m-1, k \geq 0$$

Each element of the newly generated approximation vector $v^{(k+1)}$ only depends on the constant matrix A, the constant vector f, and the previous approximation $v^{(k)}$. That property offers the possibility to perform the calculation of the vector elements of the new approximation in parallel. Furthermore, only storage space for two approximations is required since the next approximation $v^{(k+2)}$ can be stored in the no longer needed storage space of the approximation $v^{(k)}$.

Weighted Jacobi Method

The *weighted Jacobi* method is a slightly modified version of the Jacobi method. The new approximation is calculated by a weighted average of the new approximation as it is calculated by the standard Jacobi method and the previous approximation:

$$v_i^{(k+1)} := (1-\omega) v_i^{(k)} + \omega \frac{1}{a_{i,i}} \left(- \sum_{j=1, j \neq i}^{m-1} a_{i,j} v_j^{(k)} + f_i \right) \quad (3.7)$$

$$i = 1, 2, \ldots, m-1, k \geq 0$$

The factor ω dilutes or amplifies the change which takes place from one approximation to the other. In the case of $\omega = 1$ the weighted Jacobi method is identical to the standard Jacobi method.

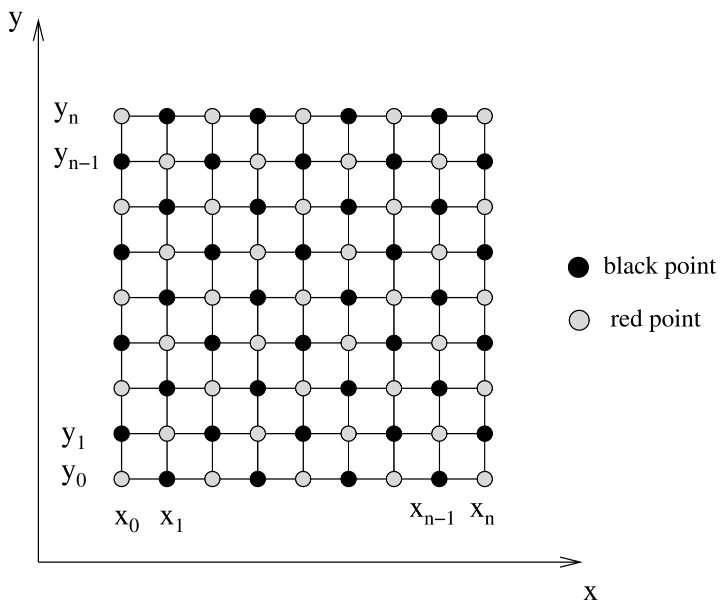

Figure 3.2: A two–dimensional grid with red–black coloring.

Gauss–Seidel Method

The two main disadvantages of the two Jacobi methods are that they require $2(m-1)$ storage space for the unknowns instead of only $m-1$ and that newly calculated information can only be used in the next step of the calculation and not as soon as it is available. The Gauss–Seidel method uses a similar approach as the Jacobi method but it uses already calculated elements of the new approximation for the calculation of other elements of the new approximation. For this purpose a new element $v_i^{(k+1)}$ of an approximation $v^{(k+1)}$ overwrites an element $v_i^{(k)}$ of the approximation $v^{(k)}$. The Gauss–Seidel method can be described in the following way:

$$v_i^{(k+1)} := -\frac{1}{a_{i,i}}\left(\sum_{j=1}^{i-1} a_{i,j} v_j^{(k+1)} + \sum_{j=i+1}^{m-1} a_{i,j} v_j^{(k)} - f_i\right) \tag{3.8}$$
$$i = 1, 2, \ldots, m-1, k \geq 0$$

The storage requirement of the Gauss–Seidel method for the unknowns is, therefore, $m-1$. However, the ordering of the nodes in the approximation vector v now plays an important role. Different orderings of the approximation elements (which are represented by grid nodes) result in different execution schemes. Instead of updating the elements in ascending order as shown above they can also be updated in descending or alternating order.

A popular ordering is the red–black ordering. A two–dimensional grid is dyed red and black as shown in Figure 3.2. First, all red nodes and then all black nodes are updated according to equation (3.8). The main advantage of the red–black ordering is that there are no data dependences between new elements which correspond to red nodes. Thus, they can be calculated in an arbitrary order since only black nodes are required for the calculation. The black nodes in turn only depend on red nodes, so that the elements of the new approximation for these nodes can be calculated in an arbitrary order as well.

This allows an easy parallelization of the red–black Gauss–Seidel method as follows. The grid is partitioned and each processor operates on one partition. In a first step the current approximation for all nodes is distributed on the parallel processors. Then, the approximation of red nodes is updated. Thereby, each processor calculates its part of the new approximation. After all red nodes have been updated, the new approximation of the red nodes is again propagated to all processors followed by the calculation of a new approximation for the black nodes. After all processors finish their work and the information of the black nodes is propagated to the other processors the process starts with the calculation of the approximation of the red nodes again. Since each processor doesn't require all grid nodes, this approach can be improved by only exchanging partition boundaries.

Successive Over Relaxation Method

The *successive over relaxation* method (SOR) is a slightly modified version of the Gauss–Seidel method. Like the weighted Jacobi method the SOR method uses a weighted average of the new and old approximation. Again, in the case of $\omega = 1$ the SOR methods is identical to the standard Gauss–Seidel method. If $0 < \omega < 1$ the impact of the change caused by the iteration on the new approximation is damped otherwise it is amplified. The calculation scheme for the SOR method is:

$$v_i^{(k+1)} := (1-\omega)v_i^{(k)} - \frac{\omega}{a_{i,i}}\left(\sum_{j=1}^{i-1} a_{i,j}v_j^{(k+1)} + \sum_{j=i+1}^{m-1} a_{i,j}v_j^{(k)} - f_i\right) \tag{3.9}$$
$$i = 1, 2, \ldots, m-1, k \geq 0$$

3.1.3 The Multigrid Idea

The representation of the problem domain with a grid leads to a discretization error since a grid usually will not represent a continuous domain correctly. Using a finer grid with a smaller grid distance will reduce the error involved with the discretization. However, it can be shown [BHM00] that the reduction of low frequency error parts decreases with decreasing grid spacing h. With decreasing h the number of grid points and consequently the amount of work to be done for one iteration will increase by $O(h^{-1})$. I.e. finer grids will require more work for one iteration and the iteration will be less efficient in respect to the error reduction. The key observation which leads to multigrid methods is that a low frequency error is of a higher frequency on grids with fewer grid points. Therefore, multigrid methods use coarser grids to reduce low frequency error parts, whereas a low discretization error and a good reduction of high frequency errors is achieved by using fine grids. There are several strategies how coarser grids can be used within multigrid methods like the correction scheme, full approximation scheme, or hierarchical multigrid. The thesis will focus on the correction scheme.

The correction scheme usually starts with a good approximation v^h of the exact solution u^h of the equation $A^h u^h = f^h$. In a first step, the high frequency error parts in the approximations are reduced. For this purpose a small number of iterations with a method like the weighted Jacobi or Gauss–Seidel method are usually sufficient. Then, the low frequency error part is reduced with a correction step on a coarser grid with grid spacing of $2h$, for example. Other grid coarsening is possible but not described in this thesis.

For the correction step, the right–hand side f^{2h} of the equation for the coarser grid is obtained by calculating the residuum r^h of the current approximation on the fine grid and restricting it to the coarser grid. The equation $A^{2h}e^{2h} = r^{2h}$ is then solved on the coarser grid and the obtained solution e^{2h} is used after interpolation to correct the current solution on the fine grid. e^{2h} is a coarse grid representation of the error of the current fine grid solution. The correction with the coarse grid representation of the error usually introduces new high frequency errors. These errors can be removed by applying additional smoothing steps after the coarse grid correction.

$$
\begin{aligned}
r^h &= f^h - A^h v^h \qquad (3.10)\\
f^{2h} &= r^{2h} = I_h^{2h} r^h \\
e^h &= I_{2h}^h e^{2h} \\
v_{new}^h &= v^h + e^h
\end{aligned}
$$

The correction on the coarser grid can in turn be obtained by a similar approach using coarser grids. In the following, three grid visiting schemes for multigrid methods will be discussed:

V–cycle Scheme

The V–cycle scheme is the basic building block for all grid visiting schemes described in this thesis. The algorithm for the V–cycle scheme is shown in Algorithm 3.1. It starts with a series of iterations on the finest grid to remove high frequency error parts. Then, it moves to successively coarser grids until the coarsest grid is reached. The solution of the coarsest grid is then interpolated to the next finer grid where it is used to correct the current approximation calculated on that grid[1]. After an additional smoothing step that approximation in turn is used to correct the current solution on the next finer grid. This process continues until the finest grid is reached. The grid visiting of the V–cycle scheme for a four level multigrid is shown in Figure 3.3.

[1]Note, that the approximations on all grids except the one on the finest grid are corrections of the error of the approximation of the next finer grid.

Algorithm 3.1 V–cycle($v_1, v_2, A^h, v^h, f^h, h$)

---

if $h = h_{max}$ **then**
  // coarsest grid is solved directly:
  solve ($A^h v^h = f^h$)
else
  // pre–smoothing
  for $i = 1$ to v_1 **do**
    relax ($A^h v^h = f^h$)
  end for
  // recursive coarse grid correction:
  $f^{2h} = I_h^{2h}(f^h - A^h v^h)$
  $v^{2h} = 0$
  generate(A^{2h});
  V–Cycle($v_1, v_2, A^{2h}, v^{2h}, f^{2h}, 2h$)
  $v^h = v^h + I_{2h}^h v^{2h}$
  // post–smoothing:
  for $i = 1$ to v_2 **do**
    relax ($A^h v^h = f^h$)
  end for
end if

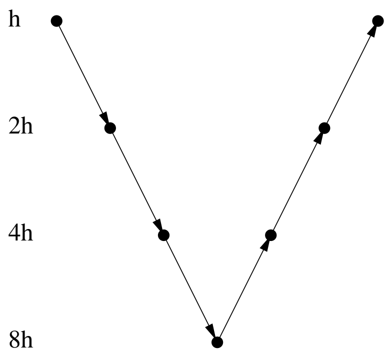

Figure 3.3: Grid visiting of the V–cycle scheme

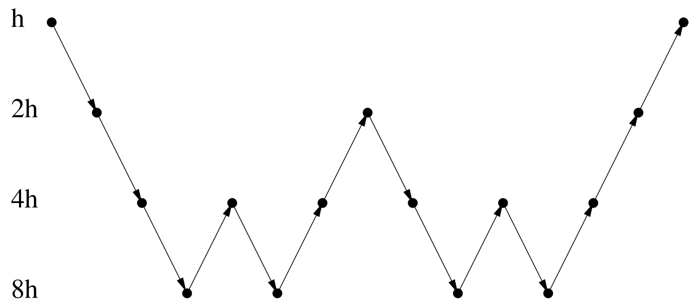

Figure 3.4: Grid visiting of the μ–cycle ($\mu = 2$) scheme

μ–cycle Scheme

The μ–cycle scheme is a trivial extension of the V–cycle scheme. The V–cycle algorithm makes exactly one recursive call to obtain the coarse grid correction. The μ–cycle algorithm illustrated in Algorithm 3.2, however, invokes itself μ times to obtain a better coarse grid correction. In practice, only $\mu = 1$ and $\mu = 2$ are used. If $\mu = 1$, the μ–cycle scheme is identical to the V–cycle scheme. The grid visiting of $\mu = 2$ for a four level multigrid method is shown in Figure 3.4.

Algorithm 3.2 μ–Cycle($v_1, v_2, \mu, A^h, v^h, f^h, h$)

  if $h = h_{max}$ **then**
    solve ($A^h v^h = f^h$)
  else
    for $i = 1$ to v_1 **do**
      relax ($A^h v^h = f^h$)
    end for
    // recursive coarse grid correction:
    $f^{2h} = I_h^{2h}(f^h - A^h v^h)$
    $v^{2h} = 0$
    generate(A^{2h});
    for $i = 1$ to μ **do**
      μ–Cycle($v_1, v_2, \mu, A^{2h}, v^{2h}, f^{2h}, 2h$)
    end for
    $v^h = v^h + I_{2h}^h v^{2h}$
    for $i = 1$ to v_2 **do**
      relax ($A^h v^h = f^h$)
    end for
  end if

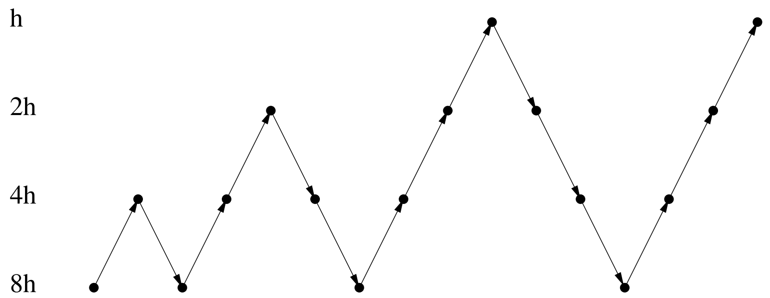

Figure 3.5: Grid visiting of the FMG scheme

Full Multigrid Scheme

So far, coarser grids have only been used to calculate corrections for the next finer grid. A coarse grid, however, can also be used to obtain a good initial guess for the finer grid. The full multigrid scheme (FMG scheme) uses that approach and combines it with the V–cycle scheme. Algorithm 3.3 shows how this is done.

The FMG scheme starts by generating a series of equations by restricting the original fine grid problem. Then, the original problem is solved on the coarsest grid with a direct solver or a sufficient amount of iteration steps to get an initial guess for the next finer grid. On the next finer grid the initial guess is then improved with a V–cycle starting and ending at the current grid to obtain a better approximation of the original problem. That approximation is then in turn used as an initial guess for the next finer grid[2]. The progress continues until the finest grid is reached. The grid schedule of the FMG scheme is illustrated in Figure 3.5.

Algorithm 3.3 FMG–V–cycle($v_1, v_2, A^{h_{min}}, v^{h_{min}}, f^{h_{min}}, h_{min}$)

generate($A^{h_{min}}, f^{h_{min}}$) *// generate A and f for every grid level*
$h = h_{max}$ *// start with coarsest grid*
solve ($A^h v^h = f^h$)
while $h > h_{min}$ **do**
  $h = \frac{h}{2}$ *// start using finer grids*
  $v^h = I_{2h}^h v^{2h}$ *// use result from coarser grid as start value*
  V–cycle($v_1, v_2, A^h, v^h, f^h, h$)
end while

[2]Note, that for the interpolation of the solution an operator I_{2h}^h of higher order is often used since the solution is not smooth in general.

3.2 Standard Multigrid Performance

It has been shown [Hac85, Bra84, TOS01] that multigrid methods are among the most efficient methods to solve partial differential equations. The work to be done in this case is linear (optimal) per unknown. These studies, however, are based on the assumption that an estimate of the number of executed operations is a good estimate for the run time of a program. With the introduction of superscalar microprocessors equipped with deep memory hierarchies other criteria like the number of memory accesses or cache hit rates have become more and more important for the estimation of the performance of an algorithm. In this section, the behavior of a "text book" multigrid method on modern superscalar microprocessors will be analyzed with a focus on run time analysis, data access analysis, and cache behavior.

3.2.1 Experimental Environment

For the experiments in this chapter the DiMEPACK library which is the result of a joint effort of the Lehrstuhl für Informatik 10, University Erlangen–Nuremberg, Germany, and the Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR–TUM), Technische Universität München, Germany will be used. DiMEPACK is a C++ library of multigrid implementations for the solution of two–dimensional partial differential equations. The library internally uses Fortran77 subroutines for the smoother and grid transfer operations. The library includes a "text book" multigrid implementation as well as cache–optimized multigrid code. In this section only the standard multigrid implementation will be used.

The red–black Gauss–Seidel smoother and the inter–grid transfer operations as well as the data locality improved routines have been implemented as part of this thesis. They build the core routines of the DiMEPACK library. The multigrid interface, the direct solver for the coarsest grid, the grid visiting schemes, and several other routines have been implemented by Markus Kowarschik (University Erlangen–Nuremberg).

The DiMEPACK library implements the V–cycle and full multigrid (FMG, nested iteration) schemes based on a coarse–grid correction scheme. Full–weighting as well as half–weighting are implemented as restriction operators. The prolongation of the coarse–grid corrections is done using bilinear interpolation. DiMEPACK uses a Gauss–Seidel/SOR smoother based on a red–black ordering of the unknowns. DiMEPACK can handle constant–coefficient problems based on discretizations using 5–point or 9–point stencils. DiMEPACK is applicable to problems on rectangular domains where different mesh widths in both space dimensions are permitted. It can handle both Dirichlet and Neumann boundary conditions. A more detailed description of the DiMEPACK library can be found in [KW01, KKRW01] and Chapter 6.

All experiments in this thesis solve the Poisson's equation $-\Delta u = \sin(2\pi x)\sin(2\pi y)$ on the unit square with equal mesh widths in both dimensions and Dirichlet boundary conditions. The values on the boundary of the domain are set to zero. In all experiments the following standard 5–point resp. 9–point stencil is used:

$$\frac{1}{h^2}\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix} \quad , \quad \frac{1}{6h^2}\begin{bmatrix} -1 & -4 & -1 \\ -4 & 20 & -4 \\ -1 & -4 & -1 \end{bmatrix}$$

Furthermore, the number of grid levels was chosen so that the coarsest grid only consists of a single unknown. If not otherwise noted, a discretization based on a 5–point stencil and the full–weighting restriction operation will be used. Unless otherwise noted, all floating point calculations will be performed with double precision floating point arithmetic.

The experiments were conducted on an Linux PC based on 700 MHz AMD Athlon (K75 core) with a peak floating point performance of 1.4 GFLOPS, another Linux PC based on a 1.5 GHz Intel Pentium 4 with a peak performance of 1.5 GFLOPS[3], a Compaq PWS 500au based on a 500 MHz Alpha 21164 with a peak performance of 1 GFLOPS, and a Compaq XP1000 based on a 500 MHz Alpha 21264 with a peak performance of 1 GFLOPS. The Athlon PC was equipped with 512 Kbyte off–chip cache whereas both Compaq machines were equipped with a 4 Mbyte off–chip cache. A detailed description of the microprocessors can be found in Chapter 2.

On the Linux machines, DiMEPACK was compiled with the GNU compiler whereas on the Compaq machines the native compiler was used. In both cases aggressive compiler optimizations were enabled.

For the experiments, the profiling tool *PCL*, *DCPI*, and *hiprof* have been used. The Performance Counter Library (*PCL*) [BM00] is a software library which provides a uniform interface to hardware performance counters of many microprocessors. The *PCL* functions can be used by application programmers to do detailed analysis on program performance and by tool writers to base their work on a common platform. *PCL* also provides a command line tool to access performance counters. The command line tool is used in the experiments. The tools *DCPI* [ABD$^+$97] and *hiprof* [Com01] are only available on the Compaq machines running Tru64 Unix. The Compaq (formerly Digital) Continuous Profiling Infrastructure (*DCPI*) is a tool set similar to *PCL* but provides an interface only for the hardware counters on the Alpha 21164 and Alpha 21264 microprocessors. Since *DCPI* provides access to more hardware counters than *PCL* on the Compaq machines, *DCPI* is used in most cases on these machines. *hiprof* is a hierarchical instruction profiler which is based on program instrumentation. *hiprof* itself is based on the *atom* instrumentation tool [Ato96, ES95] which is available for Compaq Tru64 Unix systems. *hiprof* produces a flat profile of an application that shows the execution time spent in a given procedure, and a hierarchical profile that shows the execution time spent in a given procedure and all its descendents. For the experiments only the flat profile has been used.

[3]The Pentium 4 is able to perform one floating point operation per cycle with the regular instruction set. With streaming data instructions the Pentium 4 is able to perform up to four single precision or two double precision floating point operations per cycle. It can be assumed that the gnu C++ and Fortran77 compiler which were used to compile DiMEPACK are not able to utilize these instructions. Therefore, the peak float performance of the Pentium 4 PC is 1.5 GFLOPS.

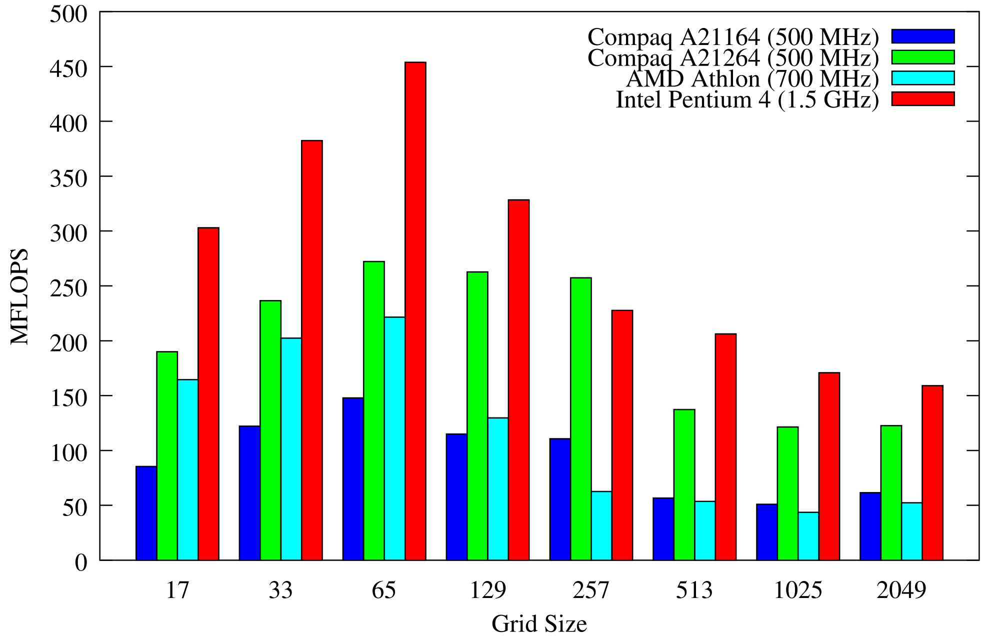

Figure 3.6: MFLOPS of a standard multigrid V–cycle scheme

3.2.2 Floating Point Performance

The key motivation of this thesis is that the machines described above deliver a peak floating point performance of one GFLOPS and beyond, but the sustained MFLOPS rates when executing a standard multigrid program are far below the theoretical performances. To illustrate this, Figure 3.6 summarizes the MFLOPS rates of DiMEPACK using two-dimensional grids of various sizes measured with *PCL*. DiMEPACK was set up to use a V–cycle scheme and full–weighting as restriction operation. The discretization was based on a 5–point stencil.

Although, the multigrid code is relatively simple and is designed to make it as easy as possible for a compiler to apply optimizations, it only exploits a fraction of the floating point peak performance of current microprocessors. Furthermore, the performance drops dramatically with growing problem size.

The Pentium 4 based machine delivers the best performance for almost all grid sizes followed by the Compaq XP1000 machine. The two other computers are clearly slower, although the Athlon is able to deliver a performance comparable to the Compaq XP1000 for smaller grid size.

The Pentium system is equipped with two data caches: an 8 Kbyte L1 data cache and a 256 Kbyte L2 cache. The L1 cache is too small to hold any multigrid data except the one for the smallest grid size. The Pentium 4 achieves the best performance of 453 MFLOPS for the 64×64 grid. The L2 cache is large enough to store the 256×256 grid completely, nevertheless, the performance deteriorates. The explanation for this is that the whole multigrid data required for solving a problem on a grid of that size is approximately twice as large and consequently doesn't fit in the L2 cache completely. Hence, the cache is able to speed up the smoother operating on the finest grid but not the whole multigrid algorithm. For the larger grid sizes not even the finest grid itself fits in the L2 cache so that another performance drop to about 200 MFLOPS can be observed for these grid sizes.

The Compaq XP1000 is equipped with a 64 Kbyte L1 data cache and a four Mbyte (off–chip) L2 cache. The L1 cache is able to store multigrid data for grids not larger than the 64×64 grid. Consequently, the best performance of approximately 270 MFLOPS is achieved for that grid size. After that the performance slightly decreases to about 260 MFLOPS. In contrast to the Pentium 4 system, however, no dramatical performance drop occurs until the grid size reaches 513×513. The L2 cache seems to be able to deliver the data fast enough to sustain the performance. The multigrid data for the three largest grids, however, is even too large for the four Mbyte L2 cache, so that the performance deteriorates to about 130 MFLOPS.

The AMD Athlon PC is also equipped with a two level cache hierarchy. Similar to the Compaq XP1000 architecture the two L1 caches are 64 Kbyte in size. Consequently, the performance trend of the multigrid on the AMD Athlon based computer is similar to the trend on the Compaq XP1000. However, the performance degradation after the data no longer fits in the L1 cache is higher and the second drop already occurs at a grid size of 257×257 due to the smaller L2 cache.

3.2.3 Analysis of the Run Time Behavior

In the following, the runtime behavior of DiMEPACK on the Compaq PWS 500au for various grid sizes will be carefully examined with the profiling tool *DCPI*. This machine is used because it provides the largest set of hardware counters and a cache hierarchy with three levels of cache which is the deepest cache hierarchy of all studied machines.

The result of the analysis is a breakdown of cycles spent for execution (Exec), nops, and different kinds of stalls (see Table 3.1). Possible causes for stalls are instruction cache misses, data cache misses (D–Cache), data translation look–aside buffer misses[4] (DTB), branch mispredictions (Branch), and register dependences (Depend).

It is interesting to note that instruction cache misses are no performance limiting issue for multigrid methods. The reason for this is that multigrid methods repeatedly execute relatively small loop bodies which easily fit in the L1 instruction cache of modern microprocessors. Consequently, instruction misses only occur when the executed multigrid component is switched, e.g. from smoother to restriction operation.

For smaller grid sizes the limiting factors are register dependences. With growing grid size, the impact of data cache misses increases and starts to dominate the runtime for the largest grids. For the largest grids cache miss stalls are responsible for over 50 per cent of all cycles of the CPU.

[4]Translation look–aside buffers (TLB) are used in conjunction with virtually addressed caches, i.e. caches whose tags are based on virtual memory addresses rather than physical addresses. A TLB caches physical address page numbers associated with virtual address pages recently in use to accelerate the address translation.

Grid Size	% of cycles used for							
	Exec	I-Cache	D-Cache	DTB	Branch	Depend	Nops	Other
17	43.3	2.2	5.1	5.2	3.3	34.4	3.7	2.8
33	38.7	1.6	7.1	5.4	1.6	39.3	3.6	2.7
65	36.4	1.0	8.1	6.0	1.0	41.6	3.6	2.3
129	32.9	0.6	10.0	8.1	0.5	41.4	3.4	3.1
257	31.5	0.3	10.8	9.8	0.3	40.9	3.3	4.1
513	15.5	0.2	27.3	34.1	0.1	18.4	1.6	2.8
1025	12.0	0.1	57.0	13.0	0.1	15.1	1.3	1.1
2049	10.5	0.2	54.3	17.3	0.1	14.1	1.3	2.2

Table 3.1: Runtime behavior of DiMEPACK on a Compaq PWS 500au.

3.2.4 Performance Limits Introduced by Instruction Distribution

Multigrid methods repeatedly process large data sets. Many load and store instructions must be executed to provide the data for these operations. However, the amount of work done once a data item is loaded into the CPU is relatively small. Table 3.2 shows the distribution of the amount of instructions executed within the multigrid program DiMEPACK. The number of executed load and store instructions is actually higher then the number of executed floating point instructions. Since load and store instructions must be executed along to multiply and add operations this creates a severe bottleneck as will be demonstrated in the following.

There a three possible strategies to execute floating point data load and store operations:

- The loads are executed with the same execution units which execute regular float operations.
- The CPU provides a separate execution unit for loads and stores.
- Load and store operations are executed within the integer unit.

If the load and store operations are executed in the same execution units as the floating point instructions, this creates a crucial limitation for the performance of multigrid methods on these architectures. For a 2049×2049 grid the floating point units are kept busy executing loads and stores for about 45 per cent of all cycles which the CPU actually spends for executing instructions, for example. Even if the remaining 55 per cent of all executed instructions are assumed to be floating point instructions and no stalls occur, the achievable performance will be limited to 55 per cent of the peak performance.

Some architectures like the AMD Athlon architecture [Pra00] provide a specialized float load and store unit which exclusively executes loads and stores for floating point data. In the case of the Athlon processor, however, there is only one load unit which has

Grid Size	% of instructions executed as				
	Integer	Float	Load	Store	Others
17	42.5	18.0	25.4	6.1	8.0
33	34.3	24.3	29.8	6.1	5.7
65	26.7	28.9	33.6	6.5	4.3
129	21.5	31.8	36.2	6.9	3.6
257	18.5	33.5	37.8	7.2	3.0
513	17.0	34.3	38.6	7.4	2.7
1025	15.9	34.0	38.4	7.6	4.1
2049	16.3	32.4	37.4	7.7	6.2

Table 3.2: Partitioning of different kinds of instruction executed for DiMEPACK.

to provide data for 2 floating point execution units. Since the amount of load and stores is higher than the amount of regular operations, the performance will be limited to the performance of the load unit.

By far the most RISC architectures execute data load and store operations within their integer units. In that case the integer units have to process the integer operations as well as all load and store operations. Thus, for a 2049×2049 grid the integer units have to process about 60 per cent of all instructions executed. This strategy will only defuse the problem if more integer than floating point units are provided by the CPU. In the case of a 2049×2049 grid a balanced system would require twice as many integer units as floating point units ($60 : 30 = 2 : 1$).

If a ratio of $1 : 1$ for integer and floating point units is assumed, no stalls of any kind occur, and in each cycle one integer and one floating point instruction can be executed, then the execution of all integer, load, and store instructions will take twice as much time as the execution of all floating point instructions. Consequently, this will limit the achievable floating point performance to 50 per cent of the peak performance. Although this observation is idealized it gives an upper bound for the achievable performance of multigrid methods. Higher performance can only be achieved if the ratio of float instructions to load and store instruction changes.

If the underlying multigrid algorithm stays unchanged, traditional cache optimization techniques which are aimed at level one or level two caches will not alleviate the problem described above. Relief can only be expected by data locality optimizations which target registers and reduce the amount of loads and stores by keeping and reusing data already within registers.

If the existence of stalls is added, e.g. triggered by data cache misses, to the exemplarily model the achievable performance may degrade further. However, stalls which slow down the execution of floating point instructions will not have such a severe impact since the total execution time spent for floating point instructions is less time than the total execution time spent for integer, load and store operations. Thus, the stall cycles are hidden since the floating point units are idle 50 per cent of the time anyway provided that a ratio

Grid Size	% of all data accesses which are satisfied by			
	L1D Cache	L2 Cache	L3 Cache	Memory
17	88.7	11.0	0.2	0.0
33	83.9	15.9	0.3	0.0
65	81.3	16.5	2.2	0.0
129	80.9	8.6	10.5	0.0
257	80.7	6.7	12.6	0.0
513	65.2	21.0	9.4	4.5
1025	48.6	39.6	6.2	5.6
2049	44.8	42.4	6.7	6.1

Table 3.3: Memory access behavior of DiMEPACK on a Compaq PWS 500au: The table shows how many per cent of all data references are served by (or loaded from) a certain level of the memory hierarchy. Thereby, the number of data references is determined by counting the number of L1 data cache accesses.

of $1:1$ is assumed for the execution units and a ratio of $60:30$ for the instructions.

3.2.5 Data Access and Cache Behavior

Since data cache misses are the dominating factor for the disappointing performance of the standard multigrid code, it seems reasonable to take a closer look at its cache behavior. Table 3.3 shows how many per cent of all data accesses are satisfied by the corresponding levels of the memory hierarchy. To obtain the data the number of L1 data cache accesses as well as the number of cache misses for each level of the memory hierarchy was measured using *DCPI*. The number of references which are satisfied by a particular level of the memory hierarchy is the difference between the number of accesses into it (misses of the memory level above it) and the number of accesses which are not satisfied by it (misses for that particular memory level). For example, the number of references satisfied by the L2 data cache is the number of L1 data cache misses minus the number of L2 data cache misses.

A different presentation of the data is shown in Table 3.4. The table shows how efficient the caches are storing data. A low miss rate indicates that a cache is very efficient. Note, that Table 3.4 and Table 3.3 are not complementary since the numbers illustrated in Table 3.3 are not hit rates except for the L1 data cache.

The analysis of the memory access behavior shows that for the 17×17 and 33×33 grids the code can access a very large fraction of the data from the L1 cache and the rest of the data from the L2 cache. For the small grid sizes the L2 data cache is a very efficient backup of the L1 data cache and provides the data with a very low miss rate. Consequently, almost no data must be loaded from the L3 cache or main memory. For the 65×65 grid we can observe that although a large fraction of the data is still delivered from the L1 cache a growing amount of accesses are fetched from the L2 cache. The

Grid Size	L1D Cache	L2 Cache	L3 Cache
17	11.3	2.0	0.2
33	16.1	1.6	0.1
65	18.7	11.6	0.1
129	19.1	55.1	0.1
257	19.3	65.3	0.3
513	34.8	39.7	32.2
1025	51.4	22.9	47.2
2049	55.2	23.2	47.9

Table 3.4: Cache miss rates (in %) of the Compaq PWS 500au running DiMEPACK. A high number indicates a bad efficiency of the particular cache level.

L2 cache delivers a large fraction of the remaining data but it isn't able to keep all the data active and the miss rate roughly increases by an order of magnitude to 11.6 per cent (see Table 3.4). Consequently, some of the data must be loaded from the L3 cache. With growing grid size this becomes even worse. For the 129×129 and 257×257 grids the L3 cache, however, is still able to deliver the data with a miss rate which is almost zero. Similarly, for grids larger then 257×257 the data no longer fits in the L3 cache and some data must be loaded from main memory. For these grid sizes we can also observe that less then 50 per cent of the data is loaded from the L1 cache and it is very inefficient in keeping the data. The efficiency of the L2 cache, however, increases again so that most of the data which isn't fetched from the L1 cache anymore can now be fetched from the L2 cache. The absolute amount of data which has to be fetched from main memory or L3 cache is relatively small. Nevertheless, this has a severe impact on the overall performance as will be demonstrated in the following:

To simplify matters, assume that the execution time is determined by the total time t_{total} spent for memory accesses. Note, that this is almost true for the largest grid. Let $t_{total} = \#access * t_{av}$. The average memory access time t_{av} calculates as follows:

$$t_{av} = t_{L1} * h_{L1} + t_{L2} * h_{L2} + t_{L3} * h_{L3} + t_{mem} * h_{mem}$$

Let h_{Lx} be the amount of data referenced from cache level x and t_{Lx} be the time required to fetch a data from cache level x. h_{mem} refers to the fraction of data which has to be loaded from main memory and t_{mem} is the time (latency) required to deliver the data from the memory to the CPU. Assume that a level one cache access requires a one cycle latency ($t_{L1} = 1c$), a level two cache access 10 cycles ($t_{L2} = 10c$), a level three cache access 50 cycles ($t_{L3} = 50c$), and a main memory access 100 cycles ($t_{mem} = 100c$).

For a 257×257 grid the average memory access time of the multigrid algorithm according to the data in Table 3.3, therefore, is $t_{av} = 0.8 * 1c + 0.07 * 10c + 0.13 * 50c = 8c$ whereas the average memory access time for a 2049×2049 grid $t_{av} = 0.45 * 1c + 0.42 *$

Module	V(1,0)	V(1,1)	V(2,1)	V(2,2)	V(4,0)	V(3,3)	V(4,4)
Smoother	48.4 %	64.9 %	73.4 %	78.4 %	78.2 %	84.7 %	88.0 %
Restriction	26.4 %	18.1 %	13.5 %	11.0 %	10.9 %	7.8 %	6.2 %
Interpolation	13.4 %	8.8 %	6.7 %	5.6 %	5.5 %	3.8 %	3.1 %
Total Multigrid	88.2 %	91.8 %	93.6 %	95.0 %	94.6 %	96.3 %	97.3 %
Init Problem	4.7 %	3.4 %	2.6 %	2.2 %	2.4 %	1.5 %	1.0 %
Init Data Structs	7.1 %	4.7 %	3.7 %	2.8 %	2.8 %	2.1 %	1.6 %
Total Overhead	11.8 %	8.2 %	6.4 %	5.0 %	5.4 %	3.7 %	2.7 %

Table 3.5: Per cent of CPU time spent in different multigrid components

$10c + 0.07 * 50c + 0.06 * 100c = 14.2c$ is already almost twice as high. The high access time for a memory access implies that a single additional per cent of memory accesses will increase the average memory access time by one cycle and consequently increase the total execution time significantly.

3.2.6 Workload Distribution Among Multigrid Components

So far, only the overall performance of the multigrid code has been analyzed. The different components of the multigrid algorithm like smoother, restriction operation, and interpolation may have different run time and memory behavior. In a first step, *hiprof* has been used to identify the most time consuming part of the multigrid program. Table 3.5 shows the per cent of work spent in the three main components of the multigrid code and the amount of overhead. The overhead summarizes the work required for the specification of the problem and initialization of the data structures. The numbers are presented in dependence to the number of applied pre– and post–smoothing steps. The column V(2,1) represents the data measured for two pre– and one post–smoothing step, for example. The analysis clearly identifies the smoother as the most time consuming part of the multigrid program. The amount of work spent in the smoother, however, depends on the amount of pre– and post smoothing steps applied within the multigrid algorithm. A typical configuration uses two pre– and two post–smoothing steps. For that configuration the smoother consumes 78.4 per cent of the runtime of the program. The second most time consuming multigrid component is the residual calculation and restriction followed by the interpolation operation.

In conjunction with the performance improvement of parts of a program, Amdahl's law [Amd67] points out that the achievable speedup for a program is determined by the fraction of time the program spends in the accelerated component of the program. Amdahl's law defines the achievable speedup for the whole program S_{total} as follows:

$$t_{new} = t_{old} * ((1 - f) + \frac{f}{S_f})$$

Grid Size	FW 5P SP	FW 5P DP	FW 9P DP	HW 5P DP	HW 9P DP
17	90.9	79.4	148.9	74.6	143.9
33	108.4	85.9	172.4	89.7	171.6
65	119.1	101.0	179.8	96.7	183.4
129	122.2	99.9	172.2	95.9	172.7
257	125.9	99.3	169.3	96.0	168.5
513	123.7	45.5	78.7	44.3	78.6
1025	115.3	37.7	62.7	36.2	63.1
2049	68.6	37.8	57.9	36.6	57.8

Table 3.6: MFLOPS of different DiMEPACK configurations.

$$S_{total} = \frac{t_{old}}{t_{new}} = \frac{1}{(1-f) + \frac{f}{S_f}} \tag{3.11}$$

$$S_{max} = \lim_{S_f \to \infty} S_{total} = \lim_{S_f \to \infty} \frac{1}{(1-f) + \frac{f}{S_f}} = \frac{1}{(1-f)} \tag{3.12}$$

t_{old} is the original runtime of the whole program. Furthermore, t_{new} is the runtime of the program after accelerating the component. f is the fraction of time spent in the component whereas S_f is the speedup achieved for the component.

Amdahl's law suggests that optimizations for multigrid codes should focus on the improvement of the smoother algorithm. The maximally achievable speedup S_{max} for the other components can be estimated with Equation 3.12. If two pre– and two post–smoothing steps are used the maximally achievable speedup by improving the restriction operation resp. the interpolation operation is 1.12 or 1.05, respectively. Therefore, optimizing them will not be very rewarding.

3.2.7 Impact of Different Multigrid Configurations

Until now, the same multigrid configuration has been used for all experiments. Using a 9–point stencil discretization or a different restriction operation might change the runtime and cache behavior of a multigrid code. To analyze the impact of different multigrid configurations the floating point performance of DiMEPACK with other configurations will be measured on the Compaq PWS 500au in the following. The result is summarized in Table 3.6.

All configurations shown in the table use double precision floating point arithmetic (DP) except the one shown in the first column which uses single precision floating point arithmetic (SP). The second column shows the MFLOPS rates for the multigrid configuration with a 5–point (5P) stencil based discretization and a full–weighting restriction operation (FW) which was used in the previous experiments for comparison. The other

Grid Size	% of all data accesses which are satisfied by			
	L1D Cache	L2 Cache	L3 Cache	Memory
17	99.4	0.5	0.1	0.0
33	94.8	5.0	0.2	0.0
65	90.0	9.5	0.5	0.0
129	89.7	7.3	3.0	0.0
257	89.4	5.4	5.2	0.0
513	88.7	5.6	5.7	0.0
1025	75.9	18.3	3.0	2.8
2049	56.0	38.4	2.3	3.3

Table 3.7: Memory access behavior of a DiMEPACK code using a discretization based on a 5–point stencil, a full–weighting restriction operation, and single precision floating point arithmetic.

columns show combinations of other configurations with a discretization based on a 9–point stencil (9P) and a half–weighting restriction operation (HW).

DiMEPACK using single precision floating point arithmetic performs significantly faster than the double precision arithmetic version especially for larger grid sizes. The speedup for the 2049×2049 grid is 1.8, almost a factor of two. Note, that a speedup measured in MFLOPS is equivalent to a speedup of the runtime in our case, since the single precision program has to exectute the same number of floating point operations as the double precision version. In fact, the compiler produces an almost identical program for both cases. Therefore, the same upper limits for the achievable floating point performance which were discussed in Section 3.2.2 apply to the single precision version as well.

The main difference between the double precision and single precision version is that the data structures of the single precision version are half as large as the double precision data structures.

As a consequence, less data must be kept in the caches, and in the case that the data is too large to fit in a cache, less data must be fetched from one of the lower levels of the memory hierarchy. Table 3.7 shows the results of an analysis of the data access behavior of the single precision DiMEPACK code. The table confirms that for the larger grids half as much data must be loaded from the L3 cache and main memory.

The reduced data set size, however, is not the only reason for this. Since a single precision word is only half as large as a double precision word a cache line will be able to store more single precision words. So, once a word is loaded into the cache twice as many words will be prefetched with the cache line and can be used without further delay. I.e. if the main memory is processed in a sequential way with an access of stride one[5], only half as many cache misses would occur with a single precision code.

[5]The stride of an access is the distance of the array elements in memory accessed within consecutive iterations of a loop.

Grid Size	% of instructions executed as				
	Integer	Float	Load	Store	Other
17	28.9	30.3	31.0	5.1	4.8
33	20.9	36.8	34.5	4.5	3.3
65	15.5	40.8	36.5	4.4	2.8
129	12.5	43.0	37.8	4.5	2.2
257	10.9	44.0	38.5	4.5	2.1
513	10.2	44.4	38.8	4.6	2.0
1025	9.9	44.0	38.6	4.7	2.7
2049	10.5	42.6	37.9	5.0	4.2

Table 3.8: Partitioning of instructions executed for a DiMEPACK code using a discretication based on a 9–point stencil and the full–weighting restriction operation.

The third and fifth column of Table 3.6 show the MFLOPS rates of DiMEPACK using a discretization based on a 9–point stencil. The numbers show that a standard multigrid algorithm with a 9–point stencil discretization performs better than one with 5–point stencil discretization. For the 9–point version the total number of data accesses as well as the total number of integer and floating point instructions is higher. However, Table 3.8 shows that the ratio of floating point to integer instructions as well as the ratio of floating point to load instructions increases. As discussed earlier too many loads and stores in a computation will limit the achievable floating point performance of multigrid codes. If the floating point data loads and stores are assumed to be executed within the integer execution units the ratio for instructions executed within the integer resp. floating point executions units is approximately $1.25 : 1$. This ratio is much better than the ratio of the 5–point stencil case. Consequently, the achievable floating point peak performance is only limited to 80 per cent of the peak performance of the CPU if a ratio of $1 : 1$ is assumed for integer and floating point instruction execution units.

The two multigrid codes with half–weighting and full–weighting restriction operation show an almost equal performance. For 5–point as well as 9–point stencil discretization, the half–weighting operator performs fewer memory references but also fewer floating point operations than the full–weighting operator. Hence, the half–weighting operator will have smaller execution time than the full–weighting operator. The ratios of floating point and load operations for the two inter–grid operations are different. The full–weighting operator performs approximately 2.5 times as many floating point instructions as the half–weighting operator but only twice as many load operations. Consequently, the floating point rates of the full–weighting operator are slightly higher than that of the half–weighting operator.

Nevertheless, the amount of time spent in the restriction operation is relatively small as showed before. So, the impact of changes in runtime and performance in the restriction operation on the total runtime and MFLOPS rate is small as well.

3.3 Cache Behavior of Red–black Gauss–Seidel

The red–black Gauss–Seidel smoother used in DiMEPACK is the most time consuming part of the multigrid method. Therefore, a standard implementation of a two-dimensional red–black Gauss–Seidel relaxation method based on a 5-point discretization of the Laplace operator, as shown in Algorithm 3.4 will be carefully analyzed in the following. The illustrated code is a simplified version of the code used within DiMEPACK. To simplify matters, Dirichlet boundaries are assumed and the handling of Neumann boundaries is removed.

Algorithm 3.4 Standard implementation of red–black Gauss–Seidel

```
double u(0 : n, 0 : n), f(0 : n, 0 : n)
for it = 1 to noIter do
  // red nodes:
  for i = 1 to n − 1 do
    for j = 1 + (i + 1)%2 to n − 1 by 2 do
      Relax( u(j, i) )
    end for
  end for
  // black nodes:
  for i = 1 to n − 1 do
    for j = 1 + i%2 to n − 1 by 2 do
      Relax( u(j, i) )
    end for
  end for
end for
```

The runtime behavior of the standard red–black Gauss–Seidel program on a Compaq PWS 500au is summarized in Table 3.9. For the smallest grid size the floating point performance is relatively high compared to the peak performance of one GFLOPS. With growing grid size the performance increases slightly to more than 450 MFLOPS. Reaching a grid size of 129×129, however, the performance dramatically drops to approximately 200 MFLOPS. For even larger grids ($> 513 \times 513$) the performance further deteriorates below 60 MFLOPS.

To detect why those performance drops occur, the program was profiled using *DCPI*. The result of the analysis is a breakdown of CPU cycles spent for execution (Exec), nops, and different kinds of stalls (see Table 3.9). Possible causes of stalls are data cache misses (Cache), data table look-aside buffer misses (DTB), branch mispredictions (Branch), and register dependences (Depend). For the smaller grid sizes the limiting factors are branch misprediction and register dependences. However, with growing grid size, the cache behavior of the algorithm seems to have an enormous impact on the runtime. Thus, for the largest grids data cache miss stalls account for more than 80 per cent of all CPU cycles.

Since data cache misses are the dominating factor for the disappointing performance

Grid Size	MFLOPS	% of cycles used for					
		Exec	Cache	DTB	Branch	Depend	Nops
17	347.0	60.7	0.3	2.6	6.7	21.1	4.5
33	354.8	59.1	10.9	7.0	4.6	11.0	5.4
65	453.9	78.8	1.4	15.7	0.1	0.0	4.2
129	205.5	43.8	6.3	47.5	0.0	0.0	2.4
257	182.9	31.9	60.6	4.2	0.0	0.0	3.3
513	175.9	31.0	60.0	4.3	0.0	0.0	2.8
1025	58.8	10.5	85.9	2.4	0.0	0.0	1.1
2049	55.9	10.1	86.5	2.4	0.0	0.0	1.1

Table 3.9: Runtime behavior of red–black Gauss–Seidel.

Grid Size	Data Set Size	% of all accesses which are satisfied by				
		±	L1 Cache	L2 Cache	L3 Cache	Memory
33	17 Kbyte	4.5	63.6	32.0	0.0	0.0
65	66 Kbyte	0.5	75.7	23.6	0.2	0.0
129	260 Kbyte	-0.2	76.1	9.3	14.8	0.0
257	1 Mbyte	5.3	55.1	25.0	14.5	0.0
513	4 Mbyte	3.9	37.7	45.2	12.4	0.8
1025	16 Mbyte	5.1	27.8	50.0	9.9	7.2
2049	64 Mbyte	4.5	30.3	45.0	13.0	7.2

Table 3.10: Memory access behavior of red–black Gauss–Seidel.

of the standard red–black Gauss–Seidel code, it seems reasonable to take a closer look at its cache behavior. Table 3.10 shows how many per cent of all array references are satisfied by the corresponding levels of the memory hierarchy. To obtain the data, the total number of array references which occur in the relaxation algorithm was counted. If all array references result in main memory references this number is equal to the number of L1 data cache accesses. Furthermore, the actual number of L1 data cache accesses as well as the number of cache misses for each level of the memory hierarchy was measured with *DCPI*.

The difference between the measured and expected number of L1 data cache accesses, is shown in column "±". Small values can be interpreted as measurement errors. Higher values, however, indicate that some of the array references are not implemented as loads or stores, but as very fast register accesses.

The analysis clearly shows that for the 33×33 and 65×65 grids the algorithm can access all of the data from the L1 or the L2 cache. However, as soon as the data does no longer fit in the L2 cache a high fraction of the data has to be fetched from the L3 cache. Similarly, for grids larger than 513×513, the data does not fit completely in the L3 cache.

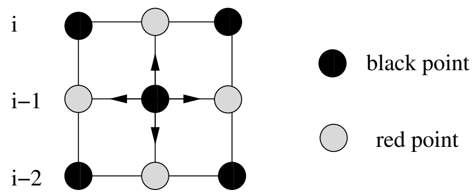

Figure 3.7: Data dependences in a red–black Gauss–Seidel algorithm.

Obviously, the memory hierarchy cannot keep all of the data close to the CPU when the size of the data grows.

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

Although the red–black Gauss–Seidel method performs global sweeps through the data set, caches can nevertheless exploit at least some temporal and spatial locality. For example, if the black node shown in the middle of Figure 3.7 is updated, the data of all the adjacent red nodes (which appear gray in the figure), the black node itself, and the corresponding value of the right–hand side of the equation (RHS) is needed. The values of the red points in lines $i-1$ and $i-2$ should be in the cache because of the update of the black points in row $i-2$. However, this is only true if at least two grid rows fit in the cache simultaneously. Also, the updated black node in line $i-1$ might be in the cache if the black node and the red node on its left side belong to the same cache line. The same argument holds for the red node in line i and the RHS value. Hence, the red node in line i, the RHS, and the black node in line $i-1$ have to be loaded from memory whenever a cache line border is crossed, which means that the data has not yet been fetched into the cache before.

Table 3.10 motivates two goals of data locality optimizations for iterative methods. First, the number of values which are loaded from the lowest levels of the memory hierarchy have to be reduced. In the case of a 1025×1025 grid, the grid data is held in main memory. The second goal is to fetch a higher fraction of the data out of one of the higher levels of the memory hierarchy, especially the registers and the L1 cache.

3.4 Multilevel Considerations

The concept of the multigrid algorithm implies that different grid sizes are visited during the computation. Possible grid visiting schemes have been introduced in Section 3.1.3.

Grid Size	% of CPU time spent for					
	Pre–relax	Restr.	Interp.	Post–relax	Total level	Total
< 513						0.4
513	0.9	0.5	0.4	0.9	2.7	3.1
1025	7.4	1.8	0.9	7.4	17.5	20.6
2049	29.4	7.7	4.2	29.4	70.7	91.3

Table 3.11: Per cent of CPU time spent on different grid levels during the execution of a V(2,2) multigrid. The finest grid involved was a 2049×2049 grid. The computation is based on a 5–point stencil discretization and a full–weighting restriction operation.

Since different grid sizes involve a different number of unknowns the work to be done in the multigrid components smoother, residual calculation, restriction, and interpolation varies dramatically.

Runtime Per Grid Level

Table 3.11 summarizes the CPU time in per cent of the total CPU time spent in different multigrid components on the different levels involved in a multigrid computation on a 2049×2049 grid using a 5–point stencil discretization, two pre– and two post–smoothing steps, and full–weighting as restriction operation.

The time spent on the grids smaller than 513×513 cannot be accounted to a single multigrid component and is negligibly small. The table clearly shows that the amount of time spent in the smoother dominates the runtime of the program. Furthermore, the calculation involved with the finest grid — smoothing the finest grid, calculating residuals on the finest grid, transferring residuals to the next coarser grid, and applying the correction from the next coarser grid to the finest grid — consumes 70.7 per cent of the total runtime whereas the computation involved with the next coarser grid involves 20.6 per cent of the total program runtime. This suggests that optimization techniques should focus on improving the performance of the multigrid components for the large grids and especially on improving the performance of the smoother component.

Residual Calculation

The multigrid methods discussed in this thesis solve a partial differential equation by smoothing the error on a fine grid and recursively calculating corrections on coarser grids. The memory and cache behavior of the smoothing step was discussed in detail in Section 3.3. Generally speaking, the smoother repeatedly performs global sweeps through typically large data sets. This type of approach offers a high potential of temporal data locality. Since CPUs, however, are not able to cache the whole data set the data locality is not exploited in usual implementations of a multigrid algorithm.

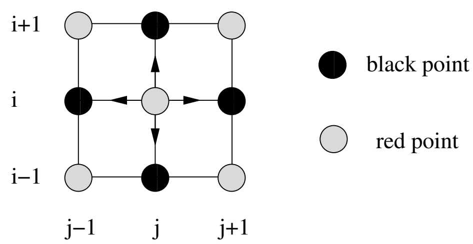

Figure 3.8: Residual calculation after red–black Gauss–Seidel smoothing.

After a smoothing step multigrid methods perform an inter–grid transfer operation. For multigrid methods which use the correction scheme this operation involves a residual calculation and the restriction of the residual to a coarser grid. The residual calculation involves another global sweep through the fine grid data structure whereas the restriction involves a global sweep through the coarser grid. Typically, these two operations are combined into one operation where the calculated residual is immediately transferred to the coarse grid. In contrast to the smoothing step, however, only one global sweep through each data structure is performed for the residual calculation and restriction operation. I.e. if the residual calculation and restriction operation is investigated separately from the smoothing steps the exploitable locality is limited to spatial locality.

The residual calculation for a fine grid point involves a computation similar to the relaxation of a node in the smoother component. If a discretization based on a 5–point stencil is assumed the calculation of the residual for the red node (i, j) shown in Figure 3.8 requires the data of the red node itself and the data of all adjacent black nodes and the corresponding value of the right–hand side of the equation. The values of the black nodes in lines i and $i-1$ should be in the cache because of the residual calculation of the red nodes $(i-1, j-1)$ and $(i-1, j+1)$. However, the locality can only be exploited if at least two grid rows fit in the cache simultaneously. Furthermore, the data for the black node $(i+1, j)$ might be cached if the data happens to be mapped to the same cache line as the black node $(i+1, j-2)$.

Restriction

To simplify matters, assume that a red–black Gauss–Seidel smoother is used during the smoothing step and the residuals of black grid points are zero. Consequently, only the residuals at red nodes have to be transferred to the next coarser grid.

Then, the half–weighting restriction operator transfers only the residual of exactly one fine grid point $(2*i, 2*j)_{fine}$ to get the right–hand side of the equation for the corresponding coarse grid point $(i, j)_{coarse}$, i.e. each coarse grid point is accessed exactly once. Thus, the data of a coarse grid point will only be in cache if it was prefetched in a cache line together with a neighboring coarse grid point.

In contrast to the half–weighting operator the full–weighting operator transfers a weighted average of the residuals of a fine grid point $(2*i, 2*j)_{fine}$ and the four neighboring red fine grid points to a coarse grid point $(i, j)_{coarse}$. There are two approaches to implement the full-weighting restriction operation. First, a sweep over the fine grid can be performed where all residuals are calculated. Once the residual of a fine grid point is determined it is immediately propagated to the coarse grid. Thus, for each fine grid point the residual of a fine grid point is calculated once but each coarse grid point is accessed five times. The second possibility to implement full–weighting is to perform a sweep over the coarse grid and calculating the residuals of the five red nodes which contribute to the coarse grid point in the process. Thus, residuals must be computed several times. Since the residual calculations are expensive, the first approach is assumed for the further discussion.

As explained above, the implementation of a full–weighting operator accesses each coarse grid point $(i, j)_{coarse}$ exactly five times. Therefore, the approach will be cache efficient if the coarse grid point $(i, j)_{coarse}$ is kept in the cache once it is accessed the first time. The accesses to the coarse grid point happen relatively close in time since they are triggered by the residual calculation of the five neighboring fine grid points $(2 * i - 1, 2 * j + 1)_{fine}$, $(2 * i + 1, 2 * j + 1)_{fine}$, $(2 * i, 2 * j)_{fine}$, $(2 * i - 1, 2 * j - 1)_{fine}$, and $(2 * i + 1, 2 * j - 1)_{fine}$. Thus, the coarse grid point will be in cache if at least three fine grid rows (the three rows $2 * j - 1$, $2 * j$, and $2 * j + 1$ with the involved fine grid nodes) and two coarse grid rows (the one with the investigated coarse grid node and the next coarse grid row) fit in the cache.

Interpolation

After the coarse grid correction is calculated it has to be propagated to the fine grid. DiMEPACK uses bilinear interpolation, i.e. the correction which is stored in a coarser grid point $(i, j)_{coarse}$ located in the interior of the grid is propagated to the finer grid point $(2 * i, 2 * j)_{fine}$ which is directly above it and to the eight neighboring nodes as illustrated in Figure 3.9. For example, the coarse grid node $(1, 1)$ propagates correction to the fine grid point $(2, 2)$ and all neighboring nodes with arrows pointing to them.

Similar to the restriction operation bilinear interpolation can be implemented in two different ways. First, one can move over the coarse grid and propagate the correction to the finer grid points. Since finer grid points (except some boundary points) get data from several coarse grid points, they are accessed several times during one sweep through the coarse grid. This algorithm is cache friendly if one grid line of the fine grid and some additional grid points fit in the cache simultaneously.

The second approach is to implement the interpolation moves sequentially through the fine grid. For a fine grid point data must be loaded from every coarse grid point which propagates a correction to that fine grid point. However, since a coarse grid point contributes to several fine grid points the coarse grid points are loaded several times. The algorithm is cache friendly if one grid line of the coarser grid and some additional grid points fit in the cache simultaneously.

The second approach is used within DiMEPACK since the first approach needs twice as much data to fit in the cache simultaneously. Furthermore, the amount of load opera-

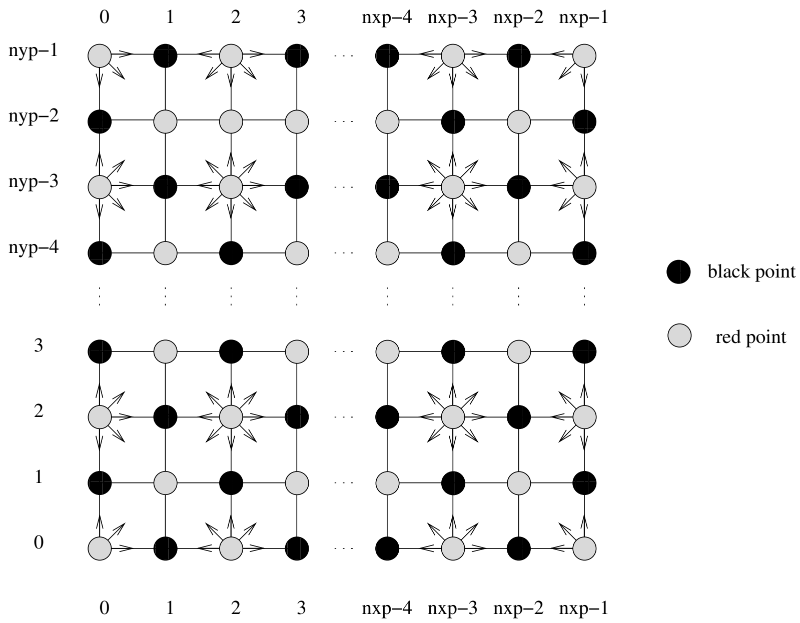

Figure 3.9: Propagation of coarse grid data to fine grid nodes.

tions to fetch data from the coarse grid can be reduced by storing data within registers and processing two fine grid lines simultaneously during one lateral move through the grid. Consider the red nodes $(0,0)$ and $(1,1)$ in Figure 3.9. Both nodes receive data from the coarse grid point directly below the fine grid point $(0,0)$. Thus, if row 0 and row 1 are processed simultaneously the data has to be loaded from main memory only once.

3.5 Summary

In this chapter, the runtime and cache behavior of a standard multigrid method in DiME-PACK is evaluated. Standard multigrid methods only exploit a small fraction of the peak performance of cache–based computer systems. Especially for large grids with data structures too large to even fit in several Mbyte large off–chip caches, the CPU is busy waiting for data to be delivered from main memory.

Much of the remaining runtime is spent for the execution of floating point instructions. However, the execution is not dominated by floating point operations but by memory operations. The typical microprocessor strategy to execute load and store instructions within integer execution units turns out to make high demands on the number of integer execution units. Therefore, the typical ratio of 1:1 for floating point execution units to integer execution units in modern RISC CPUs limits the achievable floating point performance for multigrid methods to 50 per cent of the peak floating point performance. Although many microprocessor manufacturers start to equip their CPUs with more integer execution units, CPUs with twice as many all–purpose integer units as floating point execution units are still rare.

By example of the red–black Gauss–Seidel method it has been demonstrated in this chapter that the smoother, which is by far the most time consuming part of a multigrid method, successively performs global sweeps through its data structures. Therefore, the smoother inherently has a high potential of data locality. Additionally, the smoother possesses further inherent data locality since it reuses data during the update of single nodes which was accessed while updating neighboring nodes. In general, however, the data structures are too large to fit in any cache, so that the data locality is not exploited. The consequence of this is a poor cache behavior and consequently a poor overall performance.

Similar to the smoother, the residual calculation, restriction and interpolation operations perform global sweeps through the data structures. Since only one global sweep per grid level is performed for these operations, data locality can only be exploited by combining these operations with the smoother step. The achievable speedup however, will be marginal since the percentage of the runtime spent in these functions is about 15 per cent of the total runtime for typical scenarios.

Chapter 4

Basic Data Locality Optimization Techniques

In this chapter, standard cache optimization techniques are described which in general improve instruction and data locality. Since instruction cache misses do not introduce a performance bottleneck for multigrid methods as shown in the previous chapter, this chapter will focus on data locality optimizations. Most of the optimization techniques described in this chapter, however, are not able to improve the performance of multigrid methods since data dependences prohibit their application. Nevertheless, they are required for the understanding of the more complex techniques for multigrid methods which will be proposed later in this thesis.

4.1 Introduction

Caches utilize the principle of locality that states that data (and instructions) which have been referenced recently will be accessed again in the near future. Therefore, caches buffer recently used data to speed up repeated accesses to the same data. Due to economical and technical reasons, however, caches are much smaller than main memory. As a result, very often data which was buffered in a cache for further access is replaced before it can be reused. Hardware-based techniques try to make caches "smarter" to increase the hit ratio of a cache. These techniques include higher associativity, hardware prefetching, or victim buffers, for example [HP96]. Other hardware techniques just reduce the impact of a miss by reducing the required time to reload the data like second level caches, outstanding loads, or pipelined caches.

Another approach uses software techniques to improve hit rates of caches. These techniques usually rearrange the execution or data of a program in a way that the semantics of the program stays unchanged. These software techniques can be applied by the author of a program or automatically applied by a compiler system. Software techniques which improve the performance of data accesses are called data locality optimization techniques. They can be divided into data access transformation techniques which rearrange the program execution order and data layout transformation techniques which change the layout of the program data in memory. Which optimization is appropriate for a certain program usually depends on the characteristics of the cache misses.

The misses for a cache C can be partitioned into the following three types [Hil87]:

- *compulsory misses*: The compulsory misses of a cache C are the misses which arise when a data is requested for the first time in a program. Since the data was never referenced before the data cannot be in the cache. Compulsory misses are often also called *cold* or *startup misses*.

- *capacity misses*: The capacity misses of a cache C can be determined with a fully–associative cache of similar structure as the observed cache C (i.e. same size, block size, and replacement strategy). Per definition the number of capacity misses of a cache C is equal to the number of cache misses of the fully associative cache minus the number of compulsory misses. Although this definition is precise it is not very intuitive, therefore, capacity misses are usually vaguely defined in the literature as the misses which are introduced by the limited size of a cache.

- *conflict misses*: The number of conflict misses of a cache C is the total number of cache misses minus the number of compulsory and capacity misses. Conflict misses arise from limited associativity of the cache combined with too many active memory references which are mapped to the same cache set or cache line. A memory reference is active as long as there will be further accesses to that memory location in the future. Conflict misses are also often called *interference misses*.

Optimizations for compulsory misses include hardware techniques like longer cache lines and hardware based prefetching. The prefetching technique can also be used in software to reduce the impact of cold misses. Data access transformations like *loop interchange*, *loop fusion*, or *loop blocking* can be applied to reduce the amount of capacity misses, for example. Data layout transformations rearrange the layout of data to reduce the amount of conflict misses or improve spatial locality. Data access and data layout transformations can be applied automatically by compilers or manually by the programmer to optimize the program execution time. In both cases it is important that the transformation is legal and doesn't change the semantics of the program.

A transformation is legal if for any possible data the original and new execution order produce the same result. This is very strict and in fact for many applications too strict. Slightly different results which are introduced by small rounding errors, for example, are acceptable in many cases. Some compilers allow the user to specify this. In most cases, however, dependences in the program execution order which arise by control statements like *if–else* or loop constructs as well as dependences which are introduced by variable accesses must not be violated by program transformations.

4.2 Dependence Analysis

Dependence analysis is used to determine whether a transformation can be applied without changing the semantics of a computation. A dependence is a relationship between two computations or statements that places a constraint on their execution order. The constraints must be satisfied for the code to execute correctly. There are two kinds of dependences: *control dependences* and *data dependences*. For the further reading $S_a < S_b$ will be written if a statement S_a precedes a statement S_b in a program without loops. In a program without loops a statement S_b can only depend on S_a if $S_a < S_b$.

Algorithm 4.1 Example of control and data dependences

1: $a = b$
2: **if** $a > 5$ **then**
3:     $c = b + d$
4:     $d = c$
5: **end if**
6: $c = d + 2$

A branch changes the control flow of a program depending on a certain condition. Some instructions may or may not be executed depending on the condition of the branch. Thus, there is a *control dependence* between a statement and the branch statement if the execution of the former statement depends on the condition of the branch. In Algorithm 4.1 the statement S_2 precedes statement S_3 ($S_2 < S_3$). However, S_3 is only executed if the condition $a > 5$ in statement S_2 is evaluated as true. If $S_a < S_b$ and there is a control dependence between statement S_a and S_b, then $S_a \xrightarrow{c} S_b$. In the example, there is a control dependence between S_2 and S_3 ($S_2 \xrightarrow{c} S_3$) and a control dependence between S_2 and S_4 ($S_2 \xrightarrow{c} S_4$). A more detailed description of control dependences can be found in [BGS94, Muc97].

4.2.1 Data Dependence Types

Data dependences are constraints which arise from the flow of data between statements in a program. For example statement S_1 sets the value of variable a and statement S_2 is reading it. Therefore, there is a flow of data between these two statements which constraints the execution order of these statements. There are four different types of data dependences:

- *flow dependence*: There is a flow dependence (or *true dependence*) $S_a \xrightarrow{f} S_b$ between two statement S_a and S_b, if $S_a < S_b$ and statement S_a sets a value which statement S_b later reads as input. In the example there is a flow dependence $S_3 \xrightarrow{f} S_4$ since the variable c is set by statement S_3 and used by S_4.

- *anti dependence*: There is an anti dependence $S_a \xrightarrow{a} S_b$ between two statement S_a and S_b, if $S_a < S_b$ and statement S_a uses a variable which statement S_b later sets. In

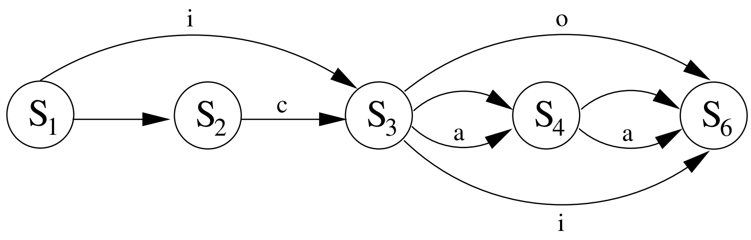

Figure 4.1: Representing dependences with a dependence graph.

the example there is an anti dependence between statement S_3 and S_4 ($S_3 \xrightarrow{a} S_4$) since S_3 reads variable d which is later set by S_4.

- *output dependence*: There is an output dependence $S_a \xrightarrow{o} S_b$ between two statement S_a and S_b, if $S_a < S_b$ and statement S_a sets the value of a variable which is also set by the statement S_b. In the example statements S_3 and S_6 both write variable c. Hence, there is an output dependence $S_3 \xrightarrow{o} S_6$.

- *input dependence*: There is an input dependence $S_a \xrightarrow{i} S_b$ between two statements S_a and S_b if both statements read the same variable. In the example both statements S_1 and S_3 read the variable b. Hence, there is an input dependence $S_1 \xrightarrow{i} S_3$. Input dependences do not apply constraints to the program execution order of two statements. However, they indicate that data is reused during program execution and a compiler might use that information for data locality optimization.

Dependences can be represented with a directed acyclic graph (DAC) called *dependence graph*. The nodes of a dependence graph represent statements. Dependences are represented by edges. Each edge is labeled to indicate the type of dependence it is representing. By convention edges which represent flow dependences are not labeled. Control flow dependences are usually not represented in a dependence graph unless such a dependence is the only edge which connects two nodes. Figure 4.1 represents the dependence graph of Algorithm 4.1.

4.2.2 Loop–carried Data Dependences

So far, the discussion was restricted to non–looping statements. In high–performance computing, however, statements which involve subscripted variables often occur within nested loops. To simplify the description of dependences with nested loops, only perfectly nested loops which are represented in a canonical form are considered in the following. That is, the index of each loop runs from 1 to n_i by 1 and only the innermost loop contains statements other than **for** statements. An example for such a loop nest is shown in Algorithm 4.2.

In non–looping and non–recursive codes each statement is executed at most once. The dependence graph described above captures all possible dependences in such codes. The

Algorithm 4.2 Loop–independent and loop–carried dependences within a loop nest.

```
1: for i_1 = 1 to 4 do
2:   for i_2 = 1 to 3 do
3:     a(i_1, i_2) = b(i_1, i_2)
4:     b(i_1, i_2) = a(i_1, i_2 - 1)
5:   end for
6: end for
```

statements within a loop nest cannot be represented individually since each statement is executed many times in general. Beside *loop–independent* dependences, *loop–dependent* (or *loop–carried*) dependences may occur. In Algorithm 4.2 there is an anti dependence between statement S_3 and S_4. This dependence is loop–independent since it occurs in each iteration independently of the surrounding loop nest. Furthermore, the code contains a loop–carried flow dependence since the variable $a(i_1, i_2 - 1)$ is used by statement S_4 in iteration (i_1, i_2) and written by statement S_3 in iteration $(i_1, i_2 - 1)$.

In codes containing no loops a statement S_b can only depend on a statement S_a ($S_a \longrightarrow S_b$) if S_a is executed before S_b ($S_a < S_b$). For loop based codes the $<$ relation must be extended. A statement $S_a[i_{1_a}, \ldots, i_{n_a}]$ which is executed in iteration $[i_{1_a}, \ldots, i_{n_a}]$ is executed before a statement $S_b[i_{1_b}, \ldots, i_{n_b}]$ which is executed in iteration $[i_{1_b}, \ldots, i_{n_b}]$ (written as $S_a[i_{1_a}, \ldots, i_{n_a}] < S_b[i_{1_b}, \ldots, i_{n_b}]$) if one of the following is true:

- S_a precedes S_b in the program and $[i_{1_a}, \ldots, i_{n_a}] \leq [i_{1_b}, \ldots, i_{n_b}]$.
- S_a and S_b are equal and $[i_{1_a}, \ldots, i_{n_a}] < [i_{1_b}, \ldots, i_{n_b}]$.
- S_a follows S_b in the program and $[i_{1_a}, \ldots, i_{n_a}] < [i_{1_b}, \ldots, i_{n_b}]$.

Consequently, there is a loop–carried flow dependence (written as $S_a[i_{1_a}, \ldots, i_{n_a}] \xrightarrow{f} S_b[i_{1_b}, \ldots, i_{n_b}]$) between two statements S_a and S_b executed in loop iteration $[i_{1_a}, \ldots, i_{n_a}]$ resp. $[i_{1_b}, \ldots, i_{n_b}]$, if $S_a[i_{1_a}, \ldots, i_{n_a}] < S_b[i_{1_b}, \ldots, i_{n_b}]$ and statement S_a sets a value which S_b later reads and $[i_{1_a}, \ldots, i_{n_a}] \neq [i_{1_b}, \ldots, i_{n_b}]$. The other types of dependences are defined analogously. Note, that if $[i_{1_a}, \ldots, i_{n_a}] = [i_{1_b}, \ldots, i_{n_b}]$ the dependence is loop–independent.

Dependence graphs are acyclic directed graphs. To represent a dependence graph for a loop, either cycles must be allowed or each execution of an individual statement within a loop nest must be represented with its own node. Thus, loop–carried dependences are usually represented with graphical visualizations of iteration spaces. Figure 4.2 shows the dependences of Algorithm 4.2, for example. Each execution of the loop body is shown with a node and loop–carried dependences between iterations are represented by edges between nodes. In the example algorithm statement S_4 within the loop nest executed in iteration $[1, 2]$ reads array element $a(1, 1)$ which is set by statement S_3 executed in iteration $[1, 1]$. So, $S_3[1, 1] < S_4[1, 2]$ and $S_3 \xrightarrow{f} S_4$. The dependence is represented as an edge between node $(1, 1)$ and $(1, 2)$ in Figure 4.2.

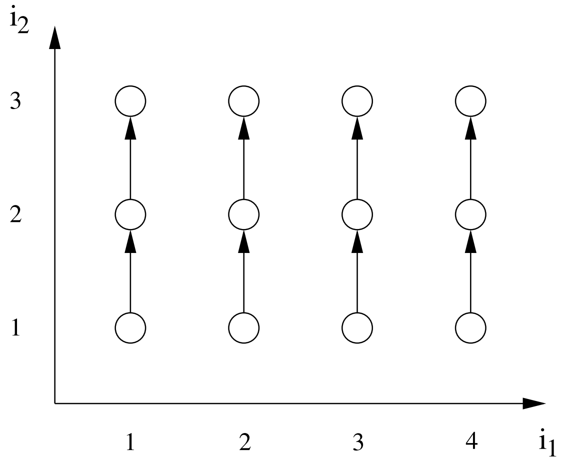

Figure 4.2: Representing loop–carried dependences.

Other representations for loop–carried dependences within loop nests are distance, direction, and dependence vectors. A description of these representations can be found in [WL91, BGS94, Muc97].

4.2.3 Dependence Testing

Many loop transformations can only be applied if no loop–carried dependences or only dependences of a particular kind are present. For a compiler a powerful dependence test is the vital basis for the loop transformation phase. The general problem of dependence testing is not computable. With some restrictions dependence testing, however, reduces to solving equations with integer coefficients for integer solutions which also satisfy given inequalities. This problem is equivalent to integer programming and known to be *NP*–complete [MHL91]. The principle of dependence testing will be demonstrated in the following by means of the Greatest Common Divisor test (*GCD* test) [Ban76, Tow76].

Algorithm 4.3 Testing dependences in loop nests.

1: **for** $i = p$ **to** q **do**
2: $\quad \ldots x(a * i + a_0) \ldots$
3: $\quad \ldots x(b * i + b_0) \ldots$
4: **end for**

To simplify matters while explaining the *GCD* test, assume a loop nest as shown in Algorithm 4.3 with two array references within the loop nest. x is a one dimensional array and p, q, a, a_0, b, and b_0 are integer constants. Both array references will access the same array element if the following equation is satisfied under the inequalities $p \leq i_1, i_2 \leq q$:

$$ai_1 + a_0 = bi_2 + b_0 \Leftrightarrow ai_1 - bi_2 = b_0 - a_0$$

For the equation $ai_1 - bi_2 = b_0 - a_0$ there is either no or an infinite number of solutions. For loop–independent dependences the equation above can be simplified to $(a-b)*i = b_0 - a_0$ since $i_1 = i_2 = i$. Two array references will be independent if $(a-b)$ does not divide $(b_0 - a_0)$. In the case of loop–carried dependences the equation has a solution if and only if $g = gcd(a, b)$ divides $(b_0 - a_0)$, i.e. if g does not divide $(b_0 - a_0)$ the two references are proven to be independent. The *GCD* test can be easily extended to loop nests and multidimensional arrays [Ban88]. Other dependence tests of which some are *NP*–complete can be found in the literature:

- Strong and weak SIV test [GKT91]
- Delta test [GKT91]
- Power test [WT90]
- Simple Loop Residue test [MHL91]
- Fourier–Motzkin test [DE73, MHL91]
- Constraint–Matrix test [Wal88]
- Omega test [PW92]

4.3 Data Access Transformations

Data access transformations are code transformations which change the order in which iterations in a loop nest are executed. The focus of these transformations is to improve data locality and register reuse. Beside improving data locality loop transformations can expose parallelism, make loop iterations vectorizable, and combinations of these. Applying a transformation will hopefully result in a performance gain on the specific target machine in use. However, it is difficult to decide which combination of transformations has to be applied to achieve a performance gain. Compilers typically use heuristics to determine whether a transformation will be effective. The loop transformation theory and algorithms found in the literature [BGS94, Muc97, Wol92] focus on transformations for perfectly nested loops. However, loop nests in scientific codes are not perfectly nested in general. Hence, enabling transformations like loop skewing, loop unrolling, loop peeling etc. are required. A description of these transformations can be found in the compiler literature [BGS94, Muc97, Wol96].

Transformations can be applied by hand or by a compiler. A programmer intuitively checks whether a transformation is legal or not. A compiler, however, needs to check after each transformation whether a loop–independent or loop–carried dependence was violated by the transformation.

In the following, a set of loop transformations will be described which focus on improving data locality for one level of the memory hierarchy. This level is typically one of the caches but may also be registers. Some of these transformations can be used to improve instruction locality as well but in some cases improving for data locality will degrade instruction locality. For example, fusing two loops can improve data locality but will make instruction locality worse if the loop body is too large after fusing to fit in the instruction cache.

4.3.1 Loop Interchange

The *loop interchange* [AK84, AK87, Wol89b] transformation reverses the order of two adjacent loops in a loop nest. Generally speaking, loop interchange works when the order of the loop execution is unimportant. Loop interchange can be generalized to *loop permutation* by allowing more than two loops to be moved at once and by not requiring them to be adjacent.

Algorithm 4.4 Loop interchange

```
double sum;
double a[n, n];

// Original loop nest:
for j = 1 to n do
    for i = 1 to n do
        sum+ = a[i, j];
    end for
end for

// Interchanged loop nest:
for i = 1 to n do
    for j = 1 to n do
        sum+ = a[i, j];
    end for
end for
```

Loop interchange can be used to enable and improve vectorization and parallelism, to improve register reuses, and to improve locality by reducing the stride of array accesses. The different targets may be conflicting. For example, increasing parallelism requires loops with no dependences to be moved outward whereas vectorization requires them to be moved inward.

Loop interchange can improve locality by reducing the stride of a computation. The stride is the distance of array elements in memory accessed within consecutive iterations of a loop. Upon a memory reference several words of an array are loaded into a cache line. Accesses with large stride only use one word per cache line with arrays bigger than the cache size. The rest of the words fetched with the cache line are evicted before they can be reused.

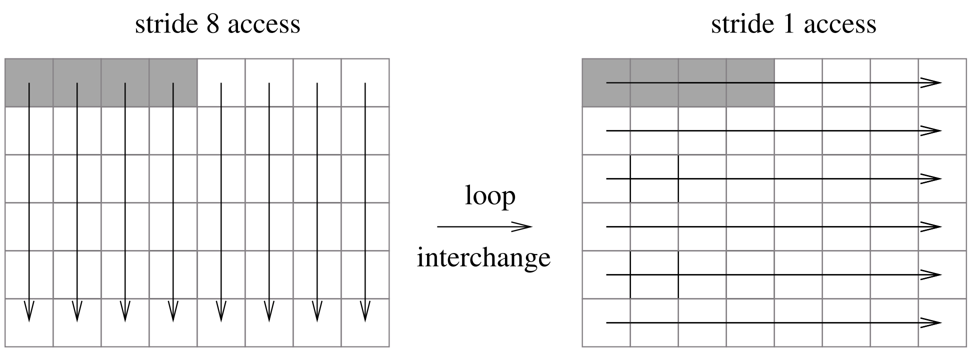

Figure 4.3: Access patterns for interchanged loop nests.

This situation is illustrated in Figure 4.3. The computation on the left side accesses all elements of a two–dimensional array of size $(6, 8)$. The array is stored in memory in a row major order, i.e. two elements of the array with second array indices which are consecutive numbers are stored adjacent in memory. The computation, however, references each row of the first column before accessing the elements of the next column. Consequently, the preloaded data in the cache line marked with grey color will not be reused if the array is too large to fit in the cache. After interchanging the loop nest as shown in Algorithm 4.4 the array accesses which have been performed contrary to the memory layout before, i.e. which have been of stride eight, are now of stride one. The stride–one array access is illustrated on the right side of Figure 4.3. Consequently, all words in the cache line are now used within executions of consecutive iterations.

4.3.2 Loop Fusion

Loop fusion [Dar99] is a transformation which takes two adjacent loops that have the same iteration space traversal and combines their bodies into a single loop, i.e. loops with the same loop bounds. The loop fusion — sometimes also called *jamming* — is the opposite operation of *loop distribution* or *loop fission* which breaks a single loop into multiple loops with the same iteration space. Loop fusion is legal as long as no flow, anti, or output dependences in the fused loop exists for which instructions from the first loop depend on instructions from the second loop.

The result of fusing two loops is that the loop body contains more instructions offering increased instruction level parallelism. Furthermore, only one loop is executed, thus, reducing the total loop overhead by a factor of two. Loop fusion also improves data locality. Assume that two consecutive loops perform global sweeps through an array like the code shown in Algorithm 4.5 and the data of the array is too large to fit completely in cache. The data which is loaded into the cache by the first loop will not completely stay in cache and the second loop will have to reload the data from main memory. If the two loops are combined with loop fusion only one global sweep will be performed through the array. Consequently, fewer cache misses will occur.

Algorithm 4.5 Loop fusion

// Original code:
for $i = 1$ **to** n **do**
    $b[i] = a[i] + 1.0;$
end for
for $i = 1$ **to** n **do**
    $c[i] = b[i] * 4.0;$
end for

// After loop fusion:
for $i = 1$ **to** n **do**
    $b[i] = a[i] + 1.0;$
    $c[i] = b[i] * 4.0;$
end for

4.3.3 Loop Blocking and Tiling

Loop blocking (also called *tiling*) is a loop transformation which increases the depth of a loop nest with depth n by adding additional loops to the loop nest. The resulting loop nest will be anything from $(n + 1)$–deep to $(2 * n)$–deep. Loop blocking is primarily used to improve data locality [GJG88, Wol89a, WL91, SL99].

The need for loop blocking is illustrated in Algorithm 4.6. Assume that the execution reads an array a with a stride of one whereas the access to array b is of stride n. Interchanging the loops won't help in that case since this will make the access to array a stride n instead.

Algorithm 4.6 Loop blocking

// Original code:
for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
        $a[i, j] = b[j, i]$
    end for
end for

// Loop blocked code:
for $ii = 1$ **to** n **by** B **do**
    for $jj = 1$ **to** n **by** B **do**
        for $i = ii$ **to** $min(ii + B - 1, n$ **do**
            for $j = jj$ **to** $min(jj + B - 1, n$ **do**
                $a[i, j] = b[j, i]$
            end for
        end for
    end for
end for

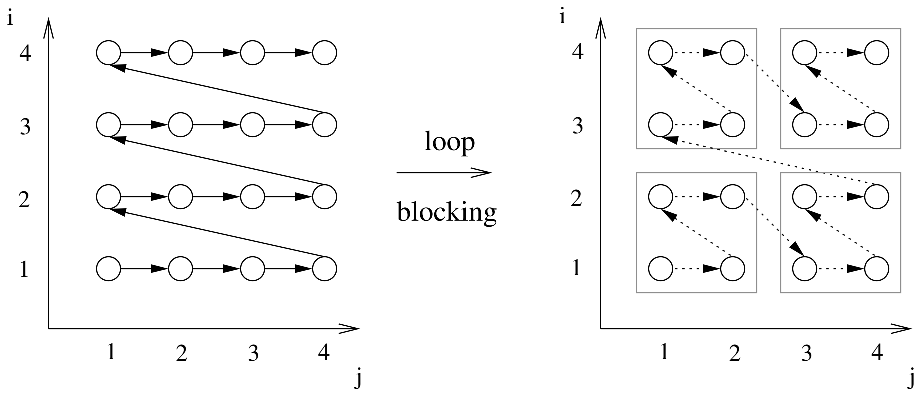

Figure 4.4: Iteration space traversal for original and blocked code.

Tiling a single loop replaces it by a pair of loops. The inner loop of the new loop nest traverses a part or block of the original iteration space with the same increment as the original loop. The outer loop traverses the original iteration space with an increment equal to the size of the block which is traversed by the inner loop. Thus, the outer loop feeds blocks of the whole iteration space to the inner loop which then executes them step by step. The change in the iteration space traversal of the blocked loop in Algorithm 4.6 is shown in Figure 4.4.

A very prominent example for the effect of the loop blocking transformation on data locality is the matrix multiplication algorithm [LRW91, BACD97, KAP97, WD98]. Algorithm 4.7 shows a text book matrix multiplication code and a blocked version of it.

Instead of operating on entire rows or columns of the arrays, the blocked algorithm operates on tiles. The tiles are (hopefully) small enough to fit in the cache and can be reused while processing the data. Thus, the amount of capacity misses is reduced. To obtain good performance for matrix multiplication the size of the tiles must be tailored to the cache size and other cache parameters like cache line size and set–associativity. In some cases, however, blocking will increase the amount of conflict misses [FST91, WL91].

4.3.4 Data Prefetching

The loop transformations discussed so far aim at reducing the capacity misses of a computation. Misses which are introduced by first time accesses are not addressed by these optimizations. *Prefetching* [VL00] allows the microprocessor to issue a data request before the computation actually requires the data. If the data is requested early enough the penalty of cold misses as well as capacity misses not covered by loop transformations can be hidden.

Many modern microprocessors nowadays implement a *prefetch* instruction which is issued as a regular instruction. The *prefetch* instruction is similar to a load, with the exception that the data is not forwarded to the CPU after it has been cached. The *prefetch*

Algorithm 4.7 Loop blocking for matrix multiplication

```
// Textbook matrix multiplication:
for i = 1 to n do
    for k = 1 to n do
        r = a[i, k];
        for j = 1 to n do
            c[i, j]+ = r * b[k, j];
        end for
    end for
end for
```

```
// Blocked matrix multiplication:
for kk = 1 to n by B do
    for jj = 1 to n by B do
        for i = 1 to n do
            for k = kk to min(kk + B - 1, n) do
                r = a[i, k];
                for j = jj to min(jj + B - 1, n) do
                    c[i, j]+ = r * b[k, j];
                end for
            end for
        end for
    end for
end for
```

instruction is often handled as a hint for the processor to load a certain data item but the fulfillment of the prefetch is not guaranteed by the CPU.

Prefetch instructions can be inserted into the code manually by the programmer or automatically by a compiler [Por89, KL91, CKP91, Mow94]. In both cases prefetching involves overhead. The prefetch instructions themselves have to be executed, i.e. pipeline slots will be filled with prefetch instructions instead of other instructions ready to be executed. Furthermore, the memory address of the prefetched data must be calculated and will be calculated again when the load operation is executed which actually fetches the data from the memory hierarchy.

Besides software prefetching hardware schemes have been proposed and implemented which add prefetching capability to a system without the need of prefetch instructions. One of the simplest hardware based prefetching schemes is sequential prefetching [Smi82]. Whenever a cache line l is accessed the cache line $l + 1$ and maybe some subsequent cache lines are prefetched. More sophisticated prefetch schemes have been invented by researchers [Jou90, JT93, CB95] but most microprocessors still implement only stride one stream detection or even no prefetching.

In general prefetching will only be successful when the data stream is predicted correctly (in hardware or by a compiler) and if there is enough space left in the cache to keep the prefetched data together with still active memory references. If the prefetched data replaces data which is still needed this will increase bus utilization, increase the overall miss rates, and memory latencies [BGK95].

4.4 Data Layout Transformations

Data access transformations have proven to be able to improve the data locality of applications by reordering the computation as shown in the previous section. However, for many applications, loop transformations alone may not be sufficient for achieving good data locality. Especially, for computations with a high degree of conflict misses loop transformations are not effective in improving performance.

Data layout transformation modify how data structures or variables are laid out in memory. These optimizations are aimed to avoid effects such as conflict misses or false sharing[1] and improve the spatial locality of a computation.

Data layout optimizations include changing base addresses of variables, modifying array sizes, transposing array dimensions, or merging of arrays. These techniques are usually applied at compile time although some optimizations can also be applied during runtime.

[1]False sharing is the result of co–location of unrelated data in the same cache unit (e.g. cache lines or pages): The data may be used by different processors such that the cache unit is shared among them but the individual data elements contained in the unit are not each referenced by all these processors.

4.4.1 Array Padding

If two arrays are accessed alternatingly as in Algorithm 4.8 and the data structures happen to be mapped to the same cache lines a high amount of conflict misses is introduced.

In the example, reading the first element of array a will load a cache line containing the first array element and possibly following array elements for further use. Provided that the first array element of array b is mapped to the same cache line as the first element of array a a read of the former element will trigger the cache to replace the elements of array a which have just been loaded. The following access to the next element of array a will no longer find that element in cache and will force the cache to reload the data and replacing the data of array b in the process. Hence, the array b elements must be reloaded and so on. Although both arrays are referenced sequentially with stride one, no reuse of data preloaded in cache lines will happen since the data is evicted immediately after it is loaded by elements of the other array. This phenomenon is called *cross interference* [LRW91] of array references.

Algorithm 4.8 Applying inter array padding.

// Original code:
double $a[1024]$
double $b[1024]$
for $i = 0$ **to** 1023 **do**
    $sum+ = a[i] * b[i]$
end for

// Code after applying inter array padding:
double $a[1024]$
double $pad[x]$
double $b[1024]$
for $i = 1$ **to** 1023 **do**
    $sum+ = a[i] * b[i]$
end for

A similar problem — called *self interference* — can occur if several rows of a multidimensional array are mapped to the same set of cache lines and the rows are accessed in an alternating fashion.

For both cases of interference array padding [BFS89, TLH90] provides a means to reduce the amount of conflict misses. *Inter array padding* inserts unused variables (pads) between two arrays to avoid cross interference between two arrays by modifying the offset of the second array so that both arrays are mapped to different cache parts.

Intra array padding on the other side inserts unused array elements between rows of a multidimensional array by increasing the leading dimension of the array, i.e. the dimension which runs faster in memory is increased by a small amount of extra elements as shown in Figure 4.5. Which dimension runs faster in memory depends on the programming language. In *Fortran77* the first dimension runs fastest in memory, for example,

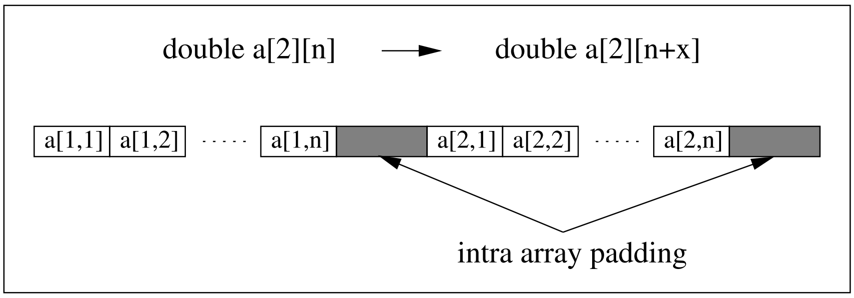

Figure 4.5: Intra array padding in a C–style language.

whereas in *C* or *C++* the last dimension runs fastest.

The size of a pad depends on the mapping scheme of the cache, cache size, cache line size, and the access pattern of the program. Typical padding sizes are multiples of the cache line size but different sizes may be used as well. Array padding is usually applied at compile time. Intra array padding can in principle be used at runtime, however, knowledge of the cache architecture is indispensable and information about the access pattern of the program will improve the quality of the selected padding size [RT98a, RT98b]. The disadvantage of array padding is that extra memory is required for pads between or within the array.

4.4.2 Array Merging

Array merging can be used to improve the spatial locality between elements of different arrays or structures. Furthermore, array merging can reduce the amount of cross interference misses in scenarios introduced in the previous section with large arrays and alternating access pattern. The array merging technique is also known as *group–and–transpose* [JE95].

Algorithm 4.9 Applying array merging.

// Original data structure:
double $a[1024]$
double $b[1024]$

// array merging using multidimensional arrays:
double $ab[1024][2]$

// array merging using structures:
struct$\{$
    double a;
    double b;
$\}$ $ab[1024]$;

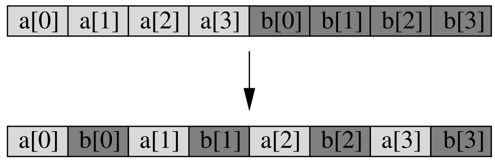

Figure 4.6: Changing data layout with array merging.

Array merging is best applied if elements of different arrays are located far apart in memory but usually accessed together. Transforming the data structures as shown in Algorithm 4.9 will change the data layout so that the elements are now contiguous in memory. The resulting data layout for two merged arrays is shown in Figure 4.6.

In some case array merging can deteriorate performance if the merged array elements are not accessed together. Cache lines will be filled with data which isn't accessed, thus, effectively reducing spatial locality.

4.4.3 Array Transpose

Array transpose [CL95] permutates the dimensions within multidimensional arrays and eventually reshapes the array as shown in Algorithm 4.10. The transformation has a similar effect as *loop interchange*.

Algorithm 4.10 Applying array transpose.

// Original data structure:
double $a[N][M]$

// Data structure after transposing:
double $a[M][N]$

4.4.4 Data Copying

In Section 4.3 the *loop blocking* or *tiling* have been introduced as a technique to reduce the amount of capacity misses. Several investigations [FST91, WL91] have shown that blocked codes suffer from a high degree of conflict misses introduced by self interference. The interference will be demonstrated by means of Figure 4.7. The figure shows a part (block) of a big array $a(i, j)$ which is to be reused by a blocked algorithm. Suppose that a direct–mapped cache is used and the two words marked with "x" are mapped to the same cache location. The periodicy of the cache mapping scheme will map the shaded words in the upper part of the block into the same cache block as the shaded words in the lower part of the block. Consequently, if the block is repeatedly accessed, the data of the upper left corner will replace the data of the lower right corner and visa versa reducing the reusable part of the block.

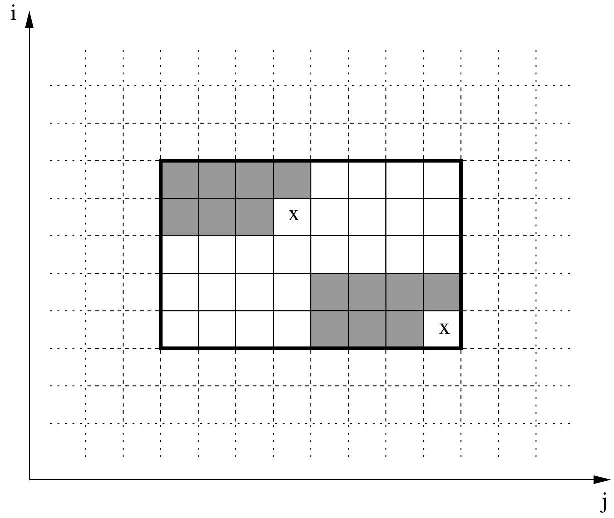

Figure 4.7: Self interference in blocked code.

Lam et al. [WL91] proposed a *data copying* technique for tiled codes to guarantee a high cache utilization for blocked algorithms. The approach copies the non–contiguous data from a block into a contiguous area. Hence, each word of the block will be mapped to its own cache location, effectively avoiding self interference within the block.

The technique, however, involves a copy operation which increases the total cost of the algorithm. In many cases the additional cost will outweigh the benefit of the data copying. Therefore, Temam et al. [TGJ93] introduced a compile time strategy to determine when to copy which data, based on an analysis of cache conflicts.

4.5 Summary

High performance on architectures with deep memory hierarchies can only be achieved by a good data locality. Data access transformations [BGS94] such as *loop interchange, loop fusion, loop blocking*, and *prefetching* are primarily used to improve cache hit rates. Most of the work concentrated on loop blocking and prefetching techniques. Data layout transformations [RT98a] such as *array padding* and *array merging* are less prominent but are useful in eliminating conflict misses and improve spatial locality.

Data locality or cache optimization techniques have been used to improve a variety of algorithms. Among them matrix multiplication algorithms [LRW91, BACD97, KAP97, WD98], linear algebra codes for dense matrices[ABB+99], FFT algorithms [FJ97], and sorting algorithms [LL97, XZK00]. Most of the approaches use domain–specific knowledge to drive transformations.

Some researchers [BACD97, WD98, FJ97] generate highly tuned cache aware packages for a certain class of algorithms based on code templates and machine parameters. These parameters are collected by benchmark programs which have to be executed on the target machine before installing the package. These programs perform an exhaustive search to identify good cache parameters for the package.

A more general approach is taken by the compiler community [WL91, KCRB98, RT98a]. They developed schemes to apply data locality optimizations automatically within a compiler pass. The optimizations within the compiler are guided by cache capacity estimates [FST91, GJG88, TFJ94] and cache miss prediction techniques [GMM99, GMM00]. Available implementations of these techniques, however, are limited to research compilers up to now. Furthermore, these data locality optimization techniques cannot be applied to complex programs as for example multigrid methods since the data dependences within the algorithm are typically to complicated to allow their application.

Chapter 5

Cache Optimization Techniques for Red–black Gauss–Seidel

The major performance bottleneck of multigrid methods in general and especially the red–black Gauss–Seidel smoother executed on computer systems equipped with deep memory hierarchies is the delay due to main memory accesses as shown in Chapter 3. Thus, data locality optimizations for the red–black Gauss–Seidel methods are promising in respect to the achievable speedup.

As shown in the previous chapter the key idea behind data locality optimizations is to reorder the data accesses so that as few of them as possible are performed between any two data references to the same memory location. With this, it is more likely that the data is not evicted from the cache and thus can be loaded from one of the higher levels of the hierarchy. However, the new access order is only valid if all data dependences are observed. Standard data locality optimizations, however, are not directly applicable to multigrid methods due to the data dependences. Therefore, this chapter proposes new program transformations which improve the data locality of the red–black Gauss–Seidel method. Thereby, the proposed transformations obey all data dependences. Consequently, the numerical results of the restructured are identical to those obtained by the original algorithms.

The new optimization techniques are able to accelerate the execution of the red–black Gauss–Seidel on currently available workstations and PCs by a multiple, especially for large grid sizes. The impact of the new techniques will be demonstrated with runtime and memory hierarchy behavior measurements as well as theoretical data locality examinations. Thereby, the data locality analysis will especially focus on aspects involved with the multilevel nature of the memory hierarchy.

The red–black Gauss–Seidel method can be directly used as iterative method to solve partial differential equations (PDEs) or as the smoother component of more complex multigrid methods. As demonstrated in Chapter 3, the smoother is by far the most time consuming component of a multigrid method. Thus, improving the performance of the smoother will result in significant speedup of the whole multigrid method as well.

5.1 Fundamental Techniques

This section will describe the principles of data locality optimization techniques for iterative methods. To simplify matters the following study will be restricted to a two–dimensional red–black Gauss–Seidel method based on a 5–point stencil discretization. Furthermore, constant coefficient problems are assumed, i.e. in addition to the right–hand side of the equation and the solution vector only five matrix coefficient for the whole grid and not five matrix coefficients for each grid point have to be stored. Extensions required for problems with variable coefficients, three–dimensional problems and discretizations based on a 9–point stencil are discussed in the following sections.

Algorithm 5.1 Standard implementation of red–black Gauss–Seidel

1: double $u(0:n,0:n)$, $f(0:n,0:n)$
2: **for** $it = 1$ **to** $noIter$ **do**
3:   *// red nodes:*
4:   **for** $i = 1$ **to** $n-1$ **do**
5:     **for** $j = 1 + (i+1)\%2$ **to** $n-1$ **by** 2 **do**
6:       Relax($u(i,j)$)
7:     **end for**
8:   **end for**
9:   *// black nodes:*
10:   **for** $i = 1$ **to** $n-1$ **do**
11:     **for** $j = 1 + i\%2$ **to** $n-1$ **by** 2 **do**
12:       Relax($u(i,j)$)
13:     **end for**
14:   **end for**
15: **end for**

The optimization process is started with a straightforward implementation of the red–black Gauss–Seidel method as illustrated in Algorithm 5.1. Note, that this version slightly differs from the version analyzed in Section 3.3[1]. The run time performance of the red–black Gauss–Seidel method on a Compaq PWS 500au and the results of a profiling experiment with *DCPI* are summarized in Table 5.1.

Although the code is rather simple the red–black Gauss–Seidel code only achieves a fraction of the floating point peak performance of 1 GFLOPS. Furthermore, the performance drops dramatically with growing problem size. The runtime of the program for large grid sizes is dominated by cache miss stalls and data TLB misses. In contrary to the analysis in Section 3.3 the TLB misses outmatch the data cache miss stalls by a factor of two.

[1]The order of the index variables in line 6 and 12 is permutated.

Grid Size	MFlops	% of cycles used for					
		Exec	Cache	DTB	Branch	Depend	Nops
17	294.9	59.9	1.8	3.1	4.2	2.5	9.0
33	242.2	49.1	18.1	5.4	3.9	1.1	8.8
65	232.2	46.8	21.7	10.6	1.1	0.8	7.3
129	150.1	30.8	38.9	12.9	1.0	1.0	6.2
257	97.9	20.1	24.3	44.9	0.4	0.5	3.9
513	60.0	12.9	27.2	52.6	0.0	0.4	2.6
1025	22.0	3.9	26.7	65.6	0.0	0.1	0.7
2049	20.4	4.1	31.0	61.4	0.0	0.0	0.6

Table 5.1: Runtime behavior of red–black Gauss–Seidel.

5.1.1 Array Transpose

The basic cause for the bad performance and the high amount of data TLB misses is an unproper memory layout of the grid data as will be explained in the following.

The first node updated by the red–black Gauss–Seidel method is the red node $(1, 1)$ followed by the red node $(1, 3)$. For the update of a red node (i, j) the data of the four adjacent black nodes $(i, j-1)$, $(i+1, j)$, $(i, j+1)$, and $(i-1, j)$ as well as the data of the red node itself and the right–hand side of the equation is required.

The data layout for the code in a Fortran77 style language is illustrated in Figure 5.1. Neighboring grid points in the same column of the grid are mapped to adjacent memory locations. Neighboring grid points within the same grid row, however, are mapped to memory locations $(n+1) * sizeof(double)$ bytes apart. Thus, two neighboring grid points within the same grid row in a double precision floating point grid of size 1025×1025 (n=1024) will be 8200 bytes apart, for example. Note, that a typical virtual memory page size for many operating systems is 8192 bytes. Hence, neighboring grid points within the same grid row may fall into different virtual memory pages. For the red–black Gauss–Seidel method this means that two of the black nodes required for an update of a red node are adjacent in memory to the updated node but the two others are $(n+1) * sizeof(double)$ memory locations apart and may fall into different virtual memory pages.

The red–black Gauss–Seidel method shown in Algorithm 5.1 starts with an update of the red nodes followed by an update of the black nodes. The red nodes in the first row are traversed first starting with the red node $(1, 1)$ followed by $(1, 3)$ and so on. After that the red nodes in row two, three and so on are updated in a similar way. If a grid size of 1025×1025 is assumed an update of a red node will involve an access to three different virtual memory pages. The first access to each page will cause a miss in the data translation look–aside buffer (TLB). The following accesses to the same virtual memory page will hit in the TLB cache. However, moving through a grid row will result in two TLB misses per update due to memory references to memory pages never accessed before. These TLB entries will be reused once the algorithm starts to update the nodes within the

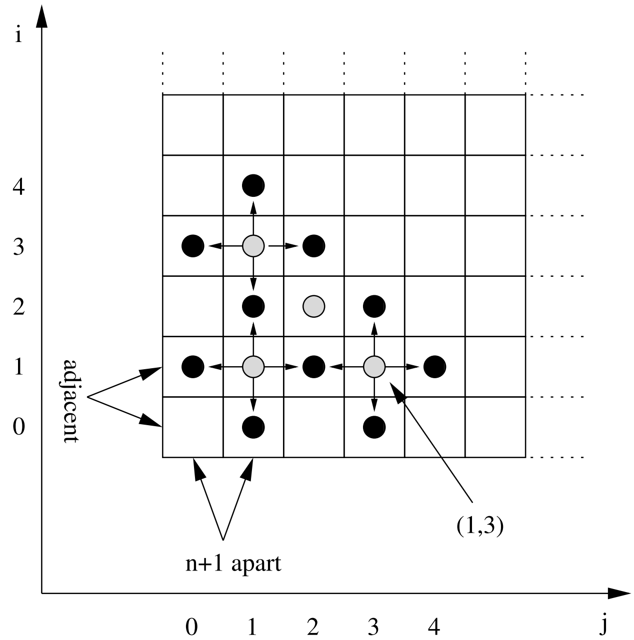

Figure 5.1: Grid memory layout before array traversal.

next grid row. However, the TLB cache is usually very small and is able to cache only a small number of entries. In our case, the amount of entries which has to fit in the TLB cache is equal to n. Thus, the size of the TLB cache will not be sufficient for larger grids. Consequently, the TLB entries for the grid elements in the lower part of the grid will be replaced a long time before they get reused and the second sweep through the next grid row will again cause two TLB misses per update. Contrary, moving through a column of the grid only requires three TLB entries for a complete sweep through the column.

This phenomenon is similar to the one which motivated the loop interchange and array traversal optimizations. Both optimizations change the traversal order of the iteration space to reduce the amount of cache misses involved with a sequence of stride–n memory accesses. Applying array traversal in our case is relatively simple since array accesses only happen in line 6 and 12 in Algorithm 5.1. A reshaping of the arrays is not required since the two arrays u and f are square.

The runtime performance of the transformed code (see Algorithm 5.2) in comparison to the original code is summarized in Table 5.2. A more detailed analysis of the runtime behavior of the improved algorithm can be found in Section 3.3. Note, that an improvement in the MFLOPS rate is equivalent to a runtime speedup of the program since the amount of executed floating point operations is unchanged. This is true for all data locality optimizations described in this thesis.

Grid Size	MFlops		Speedup
	Before	After	
17	294.9	347.0	1.2
33	242.2	354.8	1.5
65	232.2	453.9	2.9
129	150.1	205.5	1.4
257	97.9	182.9	1.9
513	60.0	63.7	1.0
1025	22.0	58.8	2.7
2049	20.4	55.9	2.7

Table 5.2: Speedup after array transpose.

Algorithm 5.2 Red–black Gauss–Seidel after array transpose

double $u(0:n, 0:n)$, $f(0:n, 0:n)$
for $it = 1$ **to** $noIter$ **do**
    // red nodes:
    for $i = 1$ **to** $n - 1$ **do**
        for $j = 1 + (i + 1)\%2$ **to** $n - 1$ **by** 2 **do**
            Relax($u(j, i)$)
        end for
    end for
    // black nodes:
    for $i = 1$ **to** $n - 1$ **do**
        for $j = 1 + i\%2$ **to** $n - 1$ **by** 2 **do**
            Relax($u(j, i)$)
        end for
    end for
end for

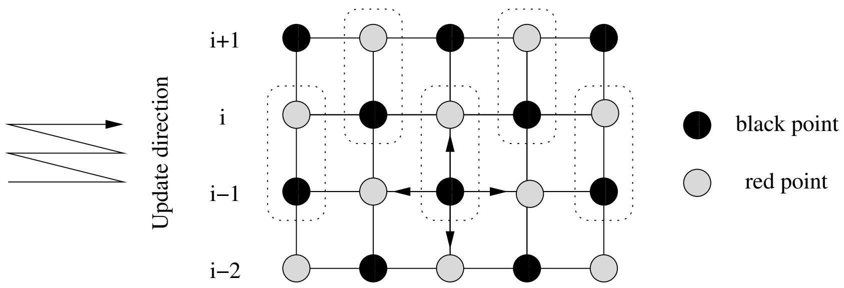

Figure 5.2: Updating red and black nodes in pairs.

5.1.2 Fusion

When implementing the standard red–black Gauss–Seidel algorithm, the usual practice is to do one complete sweep through the grid from bottom to top updating all of the red nodes and then one complete sweep through the grid updating all of the black nodes. If the grid data is too large to fit in the cache, the data of the lower part of the grid will no longer be in the cache after the first sweep because it has been replaced by the grid points belonging to the upper part of the grid. Consequently, the grid points belonging to the lower part of the grid must be reloaded from main memory which will in turn replace the cached data belonging to grid points of the upper part of the grid. As a consequence, they have to be loaded from main memory once more.

If a 5–point stencil is placed over one of the black nodes in row $i-1$ as shown in Figure 5.2, then all of the red points that are required for relaxation are up to date provided the red node directly above it in row i is up to date. Hence, no data dependences of the red–black Gauss–Seidel algorithm will be violated if the red nodes in a row i and the black nodes in row $i-1$ are updated in pairs.

This technique is called *fusion technique*. It fuses two consecutive sweeps through the grid, which update the red and black points separately, together to one sweep through the grid. However, the red nodes in the first row and the black nodes in the last row of the grid require special treatment. The red–black Gauss–Seidel method after applying the fusion technique is shown as Algorithm 5.3.

Instead of doing a sweep updating the red nodes and then a sweep updating the black nodes, just one sweep through the grid is done by the algorithm which is updating the red and black nodes together. The consequence is that the grid must be transferred from main memory to cache only once per update sweep instead of twice, provided that at least four lines of the grid fit in the cache. Three lines must fit in the cache to provide the data for the update of the black points and one additional line for the update of the red points in the line above the three lines.

An additional effect, which can improve the data locality using the fusion technique, is that the compiler might keep the values of the two updated nodes in a register, so that the update of the black node in row $i-1$ saves two load operations. Since 14 memory operations are required for the two update operations (six loads and one store for each

Algorithm 5.3 Red–black Gauss–Seidel after loop fusion

double $u(0:n,0:n)$, $f(0:n,0:n)$
for $it = 1$ **to** $noIter$ **do**
    // special handling for first grid row:
    for $j = 1$ **to** $n-1$ **by** 2 **do**
        Relax($u(j,1)$)
    end for
    // smoothing red and black nodes in pairs:
    for $i = 2$ **to** $n-1$ **do**
        for $j = 1 + (i+1)\%2$ **to** $n-1$ **by** 2 **do**
            Relax($u(j,i)$)
            Relax($u(j,i-1)$)
        end for
    end for
    // special handling for last grid row:
    for $j = 2$ **to** $n-1$ **by** 2 **do**
        Relax($u(j,n-1)$)
    end for
end for

update) about 15 per cent of all memory operations can be saved through better register usage.

Table 5.3 summarizes the memory access behavior of the red–black Gauss–Seidel algorithm after applying the fusion technique. About 20 per cent of all memory references are not satisfied by any of the cache levels nor main memory. That is, the compiler is indeed able to save two load operations as explained above by keeping the data of the two updated nodes (a red and black node) within registers. Furthermore, less data has to be loaded from the lowest (slowest) level in the memory hierarchy in which the whole grid data fits. In fact the fraction of data which has to be loaded from main memory in the case of the larger grid sizes is halved.

The optimizations are able to reduce the amount of cycles spent for data cache misses for all grid sizes except for the three smallest grid sizes (see Table 5.4). Nevertheless, a remarkable performance improvement is achieved for all grid sizes and especially for the larger grid sizes.

The fusion technique for the red–black Gauss–Seidel method in principle is based on the loop fusion data access transformation. However, loop fusion can only be applied to loops with the same loop bounds and loop increment and if no array references in the resulting loop are present with a dependence from a statement in the first loop to a statement in the second loop. In the following, a series of transformations will be developed which can be applied automatically by a compiler to perform the fusion technique to the red–black Gauss–Seidel algorithm.

The two loops with index i in the red–black Gauss–Seidel method have a similar loop

Grid Size	% of all accesses which are satisfied by				
	±	L1 Cache	L2 Cache	L3 Cache	Memory
17	19.6	80.4	0.0	0.0	0.0
33	21.0	62.8	16.1	0.0	0.0
65	18.4	70.4	10.9	0.3	0.0
129	22.3	64.7	6.1	6.8	0.0
257	21.6	42.6	30.2	5.4	0.0
513	21.8	36.7	36.8	4.3	0.4
1025	20.9	28.9	43.1	3.4	3.6
2049	20.7	29.2	34.5	12.0	3.6

Table 5.3: Memory access behavior of fused red–black Gauss–Seidel.

Grid Size	MFlops	% of cycles used for					
		Exec	Cache	DTB	Branch	Depend	Nops
17	402.6	54.3	0.5	4.5	5.4	25.8	6.4
33	458.0	60.7	11.4	3.2	2.8	10.7	9.5
65	564.2	72.7	4.8	9.2	2.8	0.0	11.0
129	363.0	50.4	36.0	4.0	2.0	0.0	8.5
257	357.0	51.6	34.0	5.5	2.0	0.0	8.7
513	196.6	34.1	52.8	5.8	1.0	0.0	4.8
1025	112.8	18.4	71.7	6.5	0.7	0.0	3.1
2049	78.6	12.6	83.5	1.4	0.5	0.0	2.1

Table 5.4: Runtime behavior of fused red–black Gauss–Seidel.

structure and would qualify for loop fusion in general. Fusing the two loops results in a pairwise update of the red and black node in a row i. As shown before the black nodes can only be updated as soon as the red node in the row directly above has been update. Hence, fusing the two loops directly violates a data dependence.

The correct procedure to fuse the two loops requires some enabling transformations. First, *loop peeling* has to be applied to the first iteration of the loop which updates the red nodes and to the last iteration of the loop which updates the black nodes. Note, that this is equivalent to the special treatment of the first and last grid row mentioned earlier. After that the loop bounds of both loops differ but fusing the two loops no longer violates a data dependence. After *normalizing* the two loops with index i, they have similar loop structure and can be fused.

The fused loop now contains two loops with index j with similar loop structure. The first loop in the fused loop relaxes all red nodes in row i before the second loop relaxes all black nodes in row $i-1$. These two loops can be fused directly. The resulting code updates red nodes in row i and black nodes in row $i-1$ in pairs while performing lateral moves through the grid. Finally, the *if*–statements which were introduced by the loop peeling transformations can be removed if the assumption is made that $n \geq 2$ and $n\%2 = 0$.

5.1.3 One–Dimensional Blocking

The optimization technique presented in the previous section improves the data locality between the global sweep through the grid which is updating all of the red nodes and the global sweep which is updating all of the black nodes. This is done by fusing the operations for the red and black update sweeps together to one operation.

If several successive red–black Gauss–Seidel relaxations have to be performed and the grid data is too large to fit entirely in the cache, the data is not reused from one iteration to the next. The *one–dimensional blocking technique* described in the following addresses that issue. Instead of performing two global sweeps through the grid which are updating the red and black nodes once each time, only one global sweep through the grid is performed updating all nodes twice.

The blocking technique is based on the data locality optimization technique *loop blocking* which was introduced in Chapter 4. Instead of operating on the whole array at once loop blocking changes the iteration traversal so that the operations are performed on small sub blocks or tiles of the whole array. The data within a tile is used as many times as possible before moving to the next block. The data of the tile is usually small enough to be kept in a faster level of the memory hierarchy.

Unfortunately, loop blocking is not directly applicable to iterative methods because of data dependences between neighboring nodes of the grid. In the case of the standard red–black Gauss–Seidel method based on a 5–point discretization of the Laplacian operator, a node of the grid cannot be touched for the second time before all of its four direct neighbors have been updated for the first time.

Consider the red node in row $i-1$ of Figure 5.3. The node can be updated for the second time provided that all neighboring black nodes have been updated once. This is

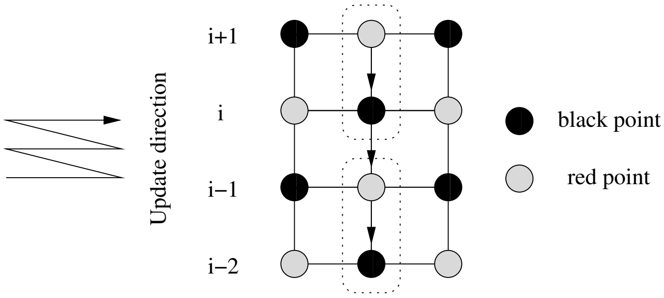

Figure 5.3: Data propagation using the 1D blocking technique.

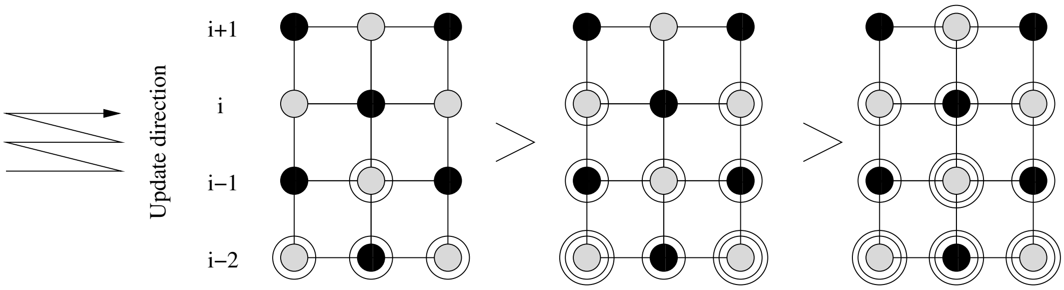

Figure 5.4: Example of blocking two red–black Gauss–Seidel sweeps. The circles around a node denote how often it has already been updated.

the case, as soon as the black node in line i directly above the red node has been updated once. This black node in turn can be updated for the first time as soon as the red node in line $i+1$ directly above it has been updated for the first time as explained earlier.

Thus, once the red node in a row $i+1$ is updated the first time the black node in row i, the red node in row $i-2$ and the black node in row $i-3$ belonging to the same column of the grid can be updated in a cascading manner. This may be done for all of the red nodes in a row $i+1$. Subsequently, the algorithm continues with relaxation in row $i+2$. Of course, some boundary handling is required in the lower and in the upper part of the grid.

An example of this proceeding is illustrated in Figure 5.4 starting in a situation where the nodes of row $i-1$ and below have been updated in pairs. The circles around the nodes indicate how often a node has been relaxed. In the next step, the red nodes in row i, the black nodes in row $i-1$, and the red nodes in row $i-2$ are update in pairs. All red nodes in row $i-2$ are now updated twice. Then, the red nodes in row $i+1$ are updated. Consequently, the black nodes in row i and $i-2$ as well as the red nodes in row $i-1$ can be updated. Now, all nodes in row $i-2$ and all red nodes in row $i-1$ are updated twice.

The one–dimensional blocking technique can be generalized to B successively performed relaxations. In that case the update of a red node in row i permits a cascading update of the nodes in row $i-1$ to $i-2*B+1$ directly below the red node. The code for the red–black Gauss–Seidel method after applying the one–dimensional blocking tech-

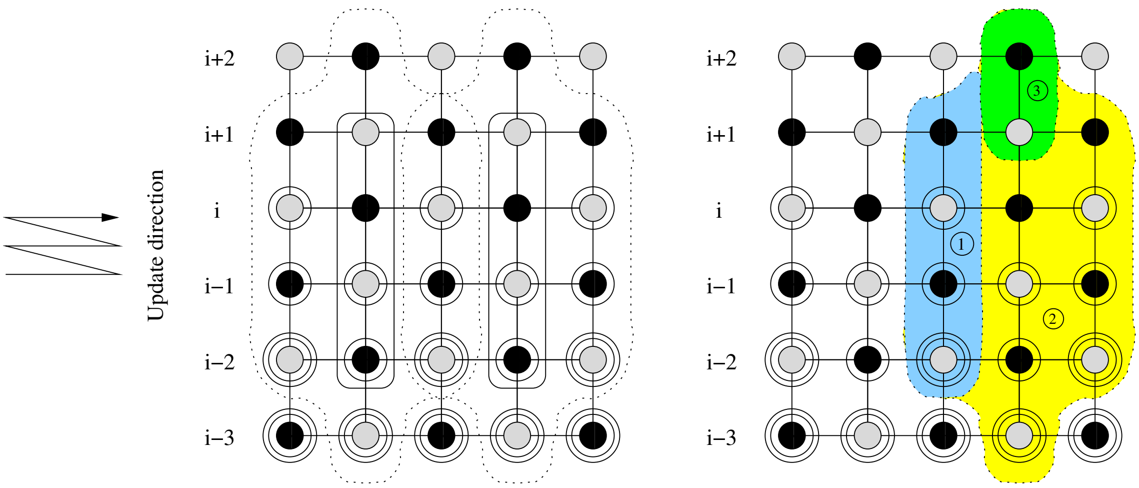

Figure 5.5: Data region classification for the one–dimensional blocking technique.

nique is represented by Algorithm 5.4[2].

The following example with $B = 2$ will illustrate the data required for the red–black Gauss–Seidel method after applying the one–dimensional blocking technique. The two groups of nodes starting in row $i + 1$ which will be updated back–to–back are highlighted with rectangles in the left side of Figure 5.5.

The data required for the update of the nodes within a rectangle are all the nodes within a dotted shape around the rectangle. The data values within the overlapping region of two adjacent dotted shapes (Area 1 in the left part of Figure 5.5) are directly reused between the cascading update of the two node groups. The size of the data in Area 1 depends on the number of blocked red–black Gauss–Seidel iterations and equals $2 * B$ nodes. Typically, a small number of successive iterations (like one to four) are performed in the multigrid context. Hence, the size of the data in Area 1 is small enough to be cached in a high level of the memory hierarchy like the L1 cache. The data in Area 2 was accessed during a previous lateral move through the grid which started cascading updates in row i. In the example the data can be loaded from cache as long as the cache is large enough to store six grid rows simultaneously. In general, $2 * B + 2$ rows of the grid have to fit in the cache simultaneously so that the data in the cache which is of a previous lateral move through the grid can be reused. Thus, the proposed technique produces a cache–aware red–black Gauss–Seidel method if at least $2 * B + 2$ grid rows fit in the cache. The remaining data required for the cascading update was never accessed before and must be loaded from main memory, unless it is prefetched in any way.

Instead of doing B successive sweeps through the grid updating each node once the

[2]Note, that the total number of iterations is assumed to be a multiple of B. If this is not the case the algorithm must be changed slightly to be semantically equivalent to the original red–black Gauss–Seidel code: after the red–black Gauss–Seidel iterations performed in the code ($noIter$ mod B) additional iterations must be performed.

Algorithm 5.4 Red–black Gauss–Seidel after one–dimensional loop blocking

```
double u(0 : n, 0 : n), f(0 : n, 0 : n)
for it = 1 to noIter/B do
  // special handling for first grid rows:
  for l = B to 1 by −1 do
    for j = 1 to n − 1 by 2 do
      Relax( u(j, 1) )
    end for
    for i = 2 to l ∗ 2 − 1 do
      for j = 1 + (i + 1)%2 to n − 1 by 2 do
        Relax( u(j, i) )
        Relax( u(j, i − 1) )
      end for
    end for
  end for
  // smoothing block wise:
  for k = B ∗ 2 to n − 1 do
    for i = k to k − B ∗ 2 + 1 by −2 do
      for j = 1 + (k + 1)%2 to n − 1 by 2 do
        Relax( u(j, i) )
        Relax( u(j, i − 1) )
      end for
    end for
  end for
  // special handling for last grid rows:
  for l = 1 to B do
    for i = (l − 1) ∗ 2 to 1 by −1  do
      for j = 1 + (i + 1)%2 to n − 1 by 2 do
        Relax( u(j, n − i) )
        Relax( u(j, n − i − 1) )
      end for
    end for
    for j = 2 to n − 1 by 2 do
      Relax( u(j, n − 1) )
    end for
  end for
end for
```

Grid Size	% of all accesses which are satisfied by				
	$\pm$	L1 Cache	L2 Cache	L3 Cache	Memory
2 relaxations combined					
17	19.4	80.6	0.0	0.0	0.0
33	20.5	77.5	2.0	0.0	0.0
65	17.8	75.2	6.8	0.2	0.0
129	21.3	57.5	18.6	2.5	0.0
257	21.5	37.1	38.7	2.7	0.0
513	21.9	30.9	39.9	3.2	0.1
1025	21.1	29.1	43.6	4.4	1.8
2049	20.8	27.5	35.4	14.3	1.9
3 relaxations combined					
17	18.8	81.2	0.0	0.0	0.0
33	20.1	78.1	1.7	0.1	0.0
65	17.4	71.4	11.0	0.2	0.0
129	21.0	50.9	25.8	2.3	0.0
257	21.2	36.7	40.2	1.9	0.0
513	21.0	35.2	41.0	2.7	0.1
1025	21.0	28.4	42.4	7.0	1.2
2049	20.9	27.2	35.6	15.1	1.2

Table 5.5: Memory access behavior of 1D blocked red–black Gauss–Seidel.

blocked red–black Gauss–Seidel method just performs one sweep through the grid updating each node B times. Hence, the grid is transferred from main memory to the cache once instead of B times provided that $2 * B + 2$ grid rows fit in the cache.

The percentage of main memory accesses needed for red–black Gauss–Seidel using the blocking technique should be equal to the percentage of main memory accesses needed for fused red–black Gauss–Seidel divided by the number of blocked iterations. Table 5.5 shows that the blocking technique does indeed reduce the number of memory accesses by a factor of two or three, respectively.

Table 5.6 summarizes the runtime behavior of the one–dimensional blocked red–black Gauss–Seidel algorithm. For the small grid sizes the optimization technique only achieves marginal improvements relative to the performance already achieved with the fused red–black Gauss–Seidel code. If three iterations are blocked the performance actually decreases. With growing grid size, however, the factor of the performance gain increases to a factor of 2.5 compared to the performance of the standard red–black Gauss–Seidel code after applying the array transpose technique for the 1025×1025 grid before it starts to decrease again. The reason for this is that the one–dimensional blocking technique expects a relatively large amount of data to fit in cache. For three blocked iterations on a 2049×2049 grid at least 130 Kbyte of data must fit in cache. The data fits easily in the

Grid Size	MFlops	% of cycles used for					
		Exec	Cache	DTB	Branch	Depend	Nops
2 relaxations combined							
17	391.3	57.8	0.2	3.5	5.8	22.2	9.0
33	489.2	64.9	5.2	2.8	3.5	10.9	11.5
65	609.9	72.4	3.1	4.2	4.2	5.3	10.7
129	404.1	58.9	25.0	5.9	2.3	0.0	10.0
257	364.6	52.7	32.1	6.2	2.0	0.0	9.0
513	255.1	41.5	45.2	5.7	1.0	0.0	6.2
1025	148.6	23.8	68.5	3.4	1.0	0.0	4.0
2049	89.6	15.1	80.5	1.5	0.6	0.0	2.4
3 relaxations combined							
17	397.4	56.3	0.2	3.3	5.0	25.1	8.1
33	444.3	61.1	10.7	3.0	2.9	10.1	10.9
65	561.0	71.5	5.8	3.4	3.7	5.3	10.6
129	404.3	57.8	24.3	5.8	2.3	1.9	9.5
257	368.2	52.4	32.4	5.9	2.0	0.0	8.9
513	268.9	43.6	41.9	5.8	1.5	0.0	6.4
1025	150.9	23.7	69.2	2.9	0.9	0.0	4.0
2049	92.5	16.1	79.5	1.5	0.6	0.0	2.6

Table 5.6: Runtime behavior of one–dimensional blocked red–black Gauss–Seidel.

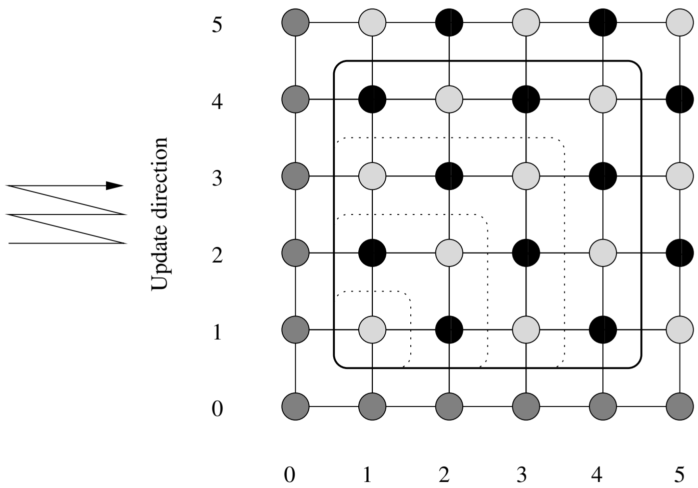

Figure 5.6: A two–dimensional subblock of the grid.

L3 cache but not in the L1 data cache or the L2 cache. In general, the fraction of time the CPU is stalled due to data delays from main memory decreases for all grid sizes. Contrary to the reduction of data cache miss stalls the fraction of time the CPU spent for branch misprediction and static stalls like register dependences increased slightly.

5.1.4 Two–Dimensional Blocking

Both of the above techniques require that a certain number of rows fit entirely in the cache. The fusion technique assumes that the cache can hold at least four rows of the grid. The one–dimensional blocking technique assumes that at least $2 * B + 2$ rows of the grid fit in the cache, if B successive sweeps through the grid are performed together. Hence, they can reduce the number of accesses to slow memory, but fail to utilize the higher levels of the memory hierarchy efficiently, in particular the registers and the L1 cache. A high utilization of the registers and the L1 cache, however, is crucial for performance in general and especially for iterative methods. In the following, *two–dimensional blocking* strategies will be proposed. As for the one–dimensional blocking technique this requires special consideration, because of the data dependences between neighboring nodes.

The key idea for standard two–dimensional loop blocking is to divide the whole array into small two–dimensional sub blocks and perform as may operations as possible for a sub block before moving to the next sub block of the array. In that manner all sub blocks are visited until the operations are performed on the whole array. The data dependences within the red–black Gauss–Seidel method, however, prevent that approach to be applied directly.

Consider the 4×4 sub block of the grid shown in Figure 5.6. A two dimensional loop blocking technique applied directly to the red–black Gauss–Seidel code starts with an update of all red–nodes in the sub block. Updating these nodes will not violate any data dependence since red nodes can be updated independently of each other. Then, all black nodes in the sub block have to be updated. The update of the black nodes in row 4 and column 4, however, is prohibited since the red nodes in row 5 and column 5 haven't been updated yet. To allow blocking, the sub block must be reduced by row 4 and column 4. Thus, the block effectively shrinks. Subsequently, the red nodes of the sub block have to be updated a second time. Since the black nodes in row 4 and column 4 haven't been updated due to data dependences only the red nodes $(2, 2)$ and $(1, 1)$ can be updated.

The one–dimensional blocking technique faces the same problem, however, only in one dimension. It solves the problem by sliding a constantly sized block from south to north through the grid. The technique virtually widens the block at the northern border to allow the update of new red nodes and consequently cascading updates of the nodes directly below the annexed nodes and narrows the block at the southern border to keep a constantly sized one–dimensional block. The same approach cannot be used directly for the two–dimensional case since the block cannot move northward and eastward at the same time. Consider the block illustrated in Figure 5.6. The block must be extended by one grid row at the northern end of the block (row 5) and by one column at the east (column 5) and shrunken by row 1 and column 1. The resulting block is no longer square since it has to accommodate the grid node $(1, 5)$ and $(5, 1)$.

The dilemma can be solved in two different ways. The first approach revisits nodes within a sub block which hasn't fully relaxed due to data dependences while the nodes within adjacent sub block are relaxed. The second approach is based on a reshaped block. Instead of a rectangular block a parallelogram is used to obey the data dependences. Furthermore, update operations within the parallelogram are performed in a line–wise manner from top to bottom. Both approaches will be described in more detail in the following.

Square Two–Dimensional Blocking

Assume that B_x is the width and B_y the height of a two–dimensional tile of the grid. The red–black Gauss–Seidel alternates relaxation sweeps for red and black nodes of the whole grid. The same approach applied to the tile will violate data dependences as described above. Nevertheless, in a sub tile of size $(B_x - 2 * B + 1) * (B_y - 2 * B + 1)$ located at the bottom left corner of the whole tile all red and black nodes can be relaxed B times without violating any data dependence provided that $B_x \geq 2 * B$ and $B_y \geq 2 * B$. The rest of the nodes will only be relaxed partially, i.e. with growing distance from the sub tile the number of relaxations decreases. Figure 5.7 illustrates the situation within a tile of size 4×4 after applying two red–black relaxations. The circles around a node represent the number of relaxations applied to the node.

The only sub tile which is completely relaxed is the sub tile in the lower left corner containing a single node. All other sub tiles (including the tile itself) are relaxed as many times as possible without violating a data dependence. The black nodes $(1, 4)$ and $(3, 4)$ have not been relaxed so far since they rely on the relaxation of the red nodes $(1, 5)$ and $(3, 5)$ which are located in the adjacent tile to the right. Hence, once the red nodes in the adjacent tile are updated the first time the black nodes $(1, 4)$ and $(3, 4)$ in the first tile

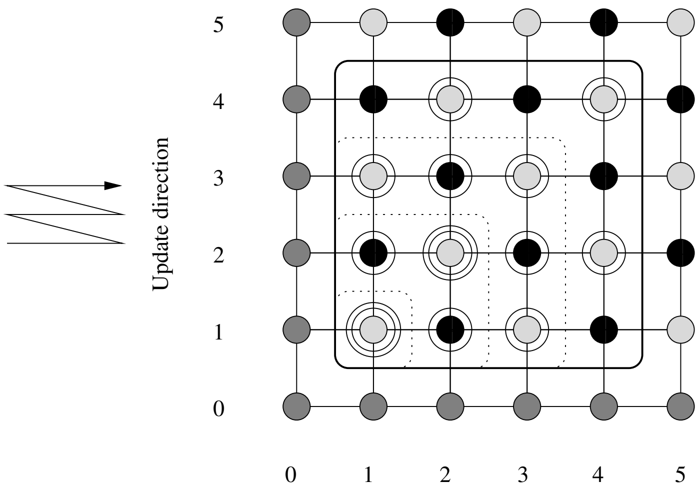

Figure 5.7: Two–dimensional blocked red–black Gauss–Seidel: Getting started

can be revisited. The supplementary relaxations of the black nodes in the first tile can be combined with the relaxation of the black nodes in the second tile (see Figure 5.8). Consequently, the red nodes $(1, 3)$ and $(2, 4)$ can be updated the second time. Again the supplementary update can be combined with the second relaxation of the red nodes in the right tile. Finally, the second update of the black node in row 1 of the left tile can be done together with the update of black nodes in the last sub tile of the right tile. Note, that the sub tile contains no black nodes in our example but may contain black nodes within larger tiles.

The situation in the right tile now is similar to the situation in the left tile when the revisiting of the tile started. The nodes in the first tile still need some more relaxations. They will occur when the tile directly above and the tile adjacent to the upper right corner are relaxed.

In general, the square blocked red–black Gauss–Seidel divides the whole grid in equally sized tiles. Tiles are visited in the same order as grid nodes in the original algorithm are visited. The blocked algorithm processes each of these tiles $2 * B$ times. Thereby, each time in turn all red resp. black nodes within the tile are updated. After each of these update sweeps the tile is moved one grid row down and one grid rows to the left to not violate any data dependences and revisit nodes within other tiles. If a tile is partially located outside the grid after moving it, the tile is narrowed until it is small enough to stay within the grid boundaries. The code for this scheme is shown in Algorithm 5.5.

The data required for a single update of a tile and the cascading updates in the moved tiles for $B = 2$ are all nodes within the dashed drawn shape around the squares in Figure 5.9. The size of the data set is independent of the grid size and only depends on the amount of blocked red–black Gauss–Seidel iterations. Although, the data set size is only $12 * B^2 + B + 1$ the data is spread over a large area of the memory, especially for large grids. This might turn out to be a disadvantage since it can involve conflict misses as well

Algorithm 5.5 Red–black Gauss–Seidel after square two–dimensional loop blocking

```
double u(0 : n, 0 : n), f(0 : n, 0 : n)
for it = 1 to noIter/B do
  // visit all tiles:
  for y_start = 1 to n + 2 * B - 1 by height do
    for x_start = 1 to n + 2 * B - 1 by width do
      // cascade sub tiles:
      for k = 0 to 2 * B - 1 do
        i_begin = y_start - k
        i_end = i_begin + height - 1
        if i_begin < 1 then
          i_begin = 1
        end if
        if i_end ≥ n then
          i_end = n - 1
        end if
        // relax a tile:
        for i = i_begin to i_end do
          j_begin = x_start - k
          j_end = j_begin + width - 1
          if j_begin < 1 then
            j_begin = 1
          end if
          j_begin = j_begin + (j_begin + i + k)%2
          if j_end ≥ n then
            j_end = n - 1
          end if
          for j = j_begin to j_end by 2 do
            Relax( u(j, i) )
          end for
        end for
      end for
    end for
  end for
end for
```

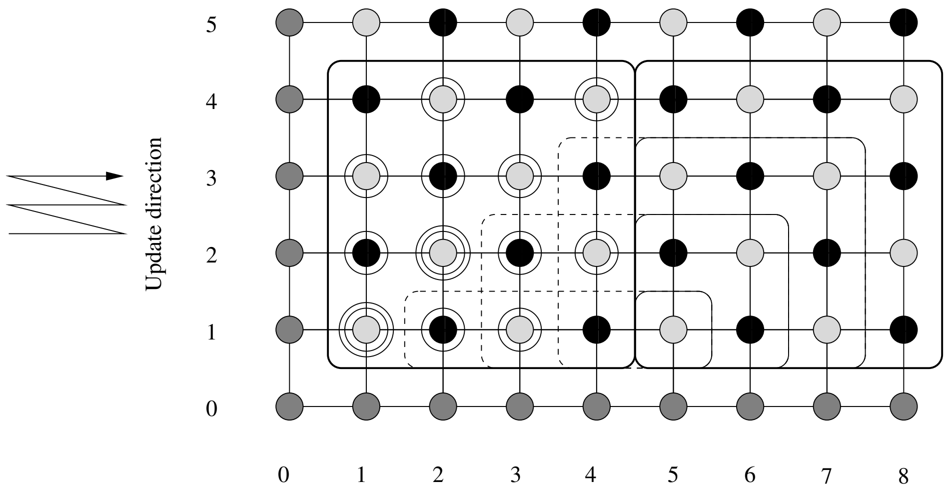

Figure 5.8: Two–dimensional blocked red–black Gauss–Seidel: Continued

as data translation look–aside buffer misses similar to the TLB miss explained in Section 5.1.1. Nevertheless, from the data locality point of view the data values in Area 1 can be reused from the update of the tile directly to the left. The data will be in the cache if the whole data required for the update — Area 1, 2 and 3 — fits in the cache simultaneously. In a multigrid context the amount of data is relatively small since usually only one to four consecutive red–black Gauss–Seidel iterations are used. Thus, the data should fit in the higher levels of the memory hierarchy like the L1 data cache. The nodes located in Area 2 can be reused from a previous lateral slide through the grid. The data, however, will be only in cache if $4 * B + 1$ grid lines fit in the cache simultaneously. This is a relatively large amount of data especially for large grids. Thus, the data will be typically cached in one of the intermediate levels of the memory hierarchy like the L2 or L3 cache. Note, that the one–dimensional blocked red–black Gauss–Seidel required only $2 * B + 2$ rows of the grid to fit in the cache simultaneously. The remaining data (Area 3) haven't been used before and must be loaded from main memory.

The square two–dimensional blocked red–black Gauss–Seidel requires more data to fit in the cache simultaneously to be a cache aware algorithm. However, in contrary two the one–dimensional blocking technique a larger fraction of the data is directly reused between the update of adjacent nodes. Furthermore, the data required for the update of a single tile (the (DTB misses)dashed area around a tile in Figure 5.10) overlaps with the data needed for the cascading update in a moved tile. In fact more then 50 per cent of the data (shaded area in Figure 5.10) is directly reused.

Skewed Two–Dimensional Blocking

The principle of the *skewed two–dimensional blocking technique* will be described in the following by means of an an example, which illustrates how all the nodes in the grid are updated twice during one global sweep. The explanation starts with the situation in

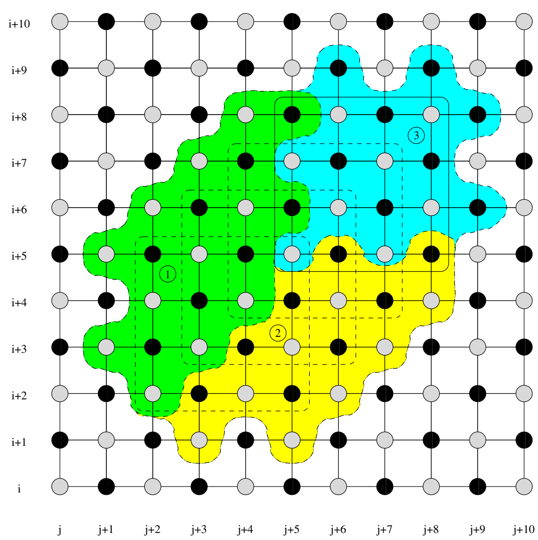

Figure 5.9: Data region classification for the square two–dimensional blocking technique.

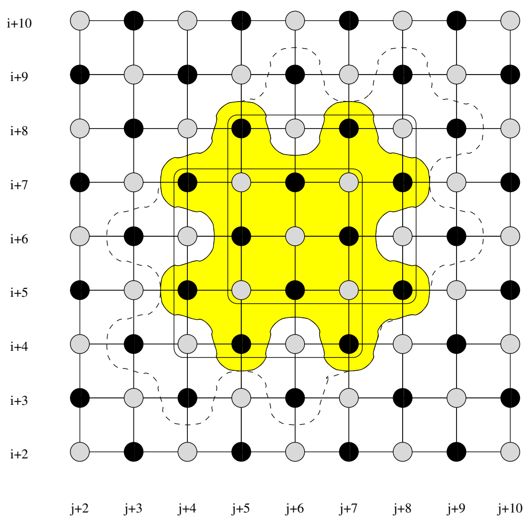

Figure 5.10: Data reused between the relaxation of cascading tiles.

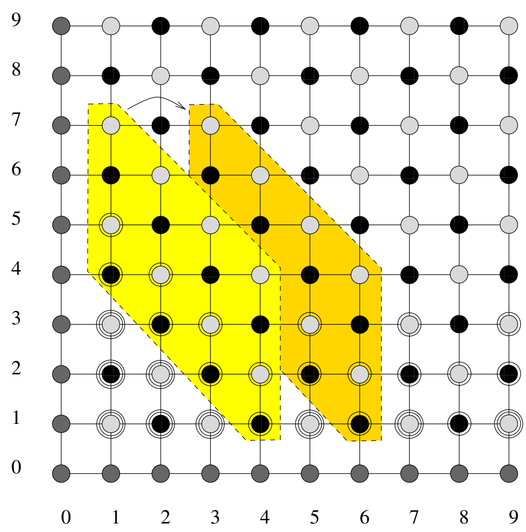

Figure 5.11: Two–dimensional blocking technique for red–black Gauss–Seidel.

Figure 5.11 where the one–dimensional blocking technique was used to create a valid initial state.

Consider the left of the parallelograms at the left of the grid. Assume that the red and the black points in its lower half have already been updated once, while the upper part stays untouched so far. The number of updates performed on each node is represented by circles around that node. The following procedure will update all nodes in the parallelogram once. For this, the diagonals are processed from top to bottom. As soon as the red points in the uppermost diagonal have been updated for the first time, the black points in the next diagonal within the parallelogram can also be updated for the first time. Then, the red and the black points on the diagonals underneath are updated. Note, that for these nodes this is now the second update. Finally, all the points in the left parallelogram have been modified. For the right parallelogram this creates the same state as initially for the left parallelogram where the nodes on the two lower diagonals have already been updated once and the two upper diagonals are still untouched. Thus, the algorithm can switch to the right parallelogram and start updating again, using the same pattern. In that fashion the parallelogram is moved through the grid until it touches the right boundary. As soon as the right boundary of the grid is reached, some extra boundary handling needs to be done before the parallelogram is moved upward by four grid lines[3]. There, some boundary handling on the left side follows, before the algorithm begins anew with the next lateral move through the grid. A pseudo code which illustrates the skewed two–dimensional blocked red–black Gauss–Seidel code is shown in Algorithm 5.6.

The data needed for an update of the grid points within one parallelogram are defined by all the nodes within a dotted drawn shape around the parallelogram (see Figure 5.11). The data values within the overlapping region of two adjacent dotted shapes (Area 1 in

[3]In general, the parallelogram has to be moved upward by $2 * B$ grid lines.

Algorithm 5.6 Red–black Gauss–Seidel after skewed two–dimensional blocking

---

for $it = 1$ **to** $noIter/B$ **do**
    Relax (bottomOfGrid())   *// special handling of first grid rows*
    for $i = 4 * B - 1$ **to** $n - 1$ **by** $2 * B$ **do**
        Relax (startTriangle(top,m,n))   *// relax initial start triangle*
        for $j = 1$ **to** $n - 2 * B - 1$ **by** 2 **do**
            Relax(parallelogram(i,j))   *// relax nodes within parallelogram*
        end for
        Relax (endTriangle(top,m,n))   *// relax finale end triangle*
    end for
    Relax (residualOfGrid())   *// relaxing residual of the grid*
    Relax (topOfGrid())   *// special handling of last grid rows*
end for

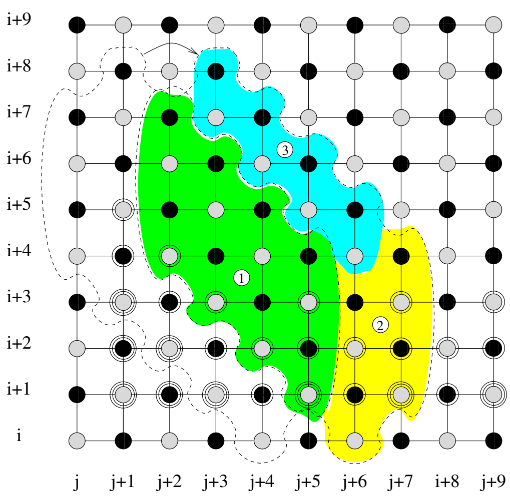

Figure 5.12: Data region classification for the skewed two–dimensional blocking technique.

Figure 5.12) are the data values which are directly reused between two parallelogram updates. The size of the data depends on the number of simultaneously performed update sweeps. Since one to four smoothing steps between the grid transfer operations are performed in typical multigrid algorithms, the size of that data should be small enough to fit at least in the L1 cache. In our example, the amount of data being reused directly is 128 Bytes (16×8 Bytes, Area 1). Area 2 can be reused from a previous slide through the grid from left to right, as long as enough grid lines can be stored in one of the levels of the memory hierarchy. Typically, the data for Area 2 is stored in one of the intermediate levels of the hierarchy. The rest of the data (Area 3), however, must be fetched from main memory.

Performance Analysis

Tables 5.7 and 5.8 reveal that the two–dimensional blocking techniques are not performing satisfactorily. For the small and for the larger grids, the floating point performance is equal or even worse than the performance of the one–dimensional blocked implementation. For some grid sizes even the standard implementation outperforms the two–dimensional blocked algorithms. Comparing the two different implementations of the two–dimensional blocking idea the skewed approach easily outperforms the square blocking technique.

The square blocking techniques suffers from a high amount of branch mispredictions as well as stalls due to register dependences especially for the small grid sizes. For the larger grids the run time is dominated by data cache stalls and DTB stalls, however, branch mispredictions and register dependences have still a noticeable impact on the performance. If more red–black Gauss–Seidel iterations are blocked the fraction of cycles spent due to branch mispredictions and register dependences decreases. Consequently, the performance for the small grid sizes increases significantly. The performance for the large grids, however, is not affected because of an increased amount of data cache miss stalls.

The skewed two–dimensional blocking technique outperforms the square blocking approach for the smaller grid sizes because of less branch mispredictions and register dependences. For the larger grids, however, the performance of both approaches is limited by data cache stalls. Again, blocking more iterations is not improving performance.

Both approaches require complicated loop structures which make other loop transformations like *loop unrolling* or *software pipelining* harder. Thus, the compiler is not able to fully optimize the schedule of assembler instructions to avoid register dependences and branch mispredictions. However, this does not explain the high fraction of time spent for data cache misses since both techniques are designed to reduce the number of data cache misses.

The memory access behavior analysis for both approaches is summarized in Table 5.9 and Table 5.10. Both techniques fail to keep a reasonable fraction of the data within the L1 cache although both techniques should be able to reuse a large fraction of the data from the L1 cache as demonstrated above. Even the L2 cache is not able to cache a reasonable

Grid Size	MFlops	% of cycles used for					
		Exec	Cache	DTB	Branch	Depend	Nops
2 relaxations combined							
17	111.3	44.0	0.1	3.5	5.9	40.9	1.1
33	105.0	38.9	10.6	3.5	4.0	38.7	1.1
65	110.2	40.3	4.0	3.8	4.3	41.8	2.2
129	108.2	39.4	12.0	3.4	3.8	37.1	0.0
257	102.2	37.0	13.6	5.8	3.6	35.0	2.1
513	82.9	29.0	29.2	7.1	2.8	27.9	0.4
1025	55.0	15.6	32.6	33.1	1.4	14.8	0.6
2049	38.2	13.6	57.3	13.9	1.4	11.8	0.9
3 relaxations combined							
17	118.0	40.5	0.3	1.9	6.9	43.7	1.2
33	112.5	35.3	10.5	1.7	5.2	41.6	1.1
65	120.4	37.7	2.5	1.8	5.6	46.0	2.4
129	117.9	36.4	11.6	1.5	4.3	40.3	0.0
257	113.3	35.3	12.8	2.6	4.2	39.1	2.3
513	95.7	28.6	27.0	3.6	3.7	32.2	0.3
1025	60.4	16.5	44.0	15.1	2.7	18.4	0.7
2049	37.0	12.2	68.1	3.9	2.0	12.0	0.9
4 relaxations combined							
17	196.6	53.1	0.3	1.8	5.4	30.5	3.7
33	196.8	49.1	22.1	1.1	3.1	15.7	4.0
65	216.7	56.6	12.4	9.1	1.9	7.1	7.1
129	231.7	58.9	18.1	4.9	1.3	3.1	7.2
257	207.7	50.7	28.9	2.4	0.9	1.5	7.0
513	157.0	37.2	41.3	11.4	0.7	0.6	4.9
1025	71.1	18.1	68.2	8.6	0.2	0.1	2.7
2049	36.2	9.9	84.2	2.9	0.2	0.1	1.7

Table 5.7: Runtime behavior of square two–dimensional blocked red–black Gauss–Seidel.

Grid Size	MFlops	% of cycles used for					
		Exec	Cache	DTB	Branch	Depend	Nops
2 relaxations combined							
17	393.2	60.6	0.5	1.1	2.6	24.9	13.1
33	340.4	46.2	31.6	1.2	1.5	5.9	9.4
65	361.2	47.5	33.2	1.4	0.5	3.4	10.2
129	314.4	43.3	43.5	0.8	0.3	1.4	8.8
257	302.1	39.8	47.8	0.6	0.0	1.1	8.5
513	255.1	33.9	54.5	0.6	0.1	1.3	7.6
1025	91.4	13.1	81.8	0.3	0.0	0.4	3.1
2049	45.5	6.6	90.8	0.2	0.1	0.3	1.5
3 relaxations combined							
17	368.6	51.6	0.4	1.8	1.9	29.4	8.2
33	304.7	43.5	12.6	0.4	4.0	28.9	2.3
65	314.6	45.0	32.5	0.9	0.3	7.0	9.1
129	309.5	42.6	36.5	0.7	0.2	5.1	10.6
257	284.8	38.8	41.7	1.3	0.1	4.8	9.6
513	231.4	33.1	50.3	1.0	0.1	4.8	7.0
1025	82.5	12.3	80.2	1.2	0.1	1.6	3.0
2049	47.8	7.3	87.4	1.6	0.0	1.0	1.8
4 relaxations combined							
17	357.5	52.8	0.4	1.5	1.3	26.2	10.0
33	276.8	42.3	14.4	0.3	5.0	28.9	2.2
65	276.7	42.3	13.6	0.4	5.1	28.6	0.9
129	269.5	41.0	15.6	0.4	1.7	32.7	0.0
257	250.8	39.4	17.6	0.7	2.1	30.7	0.0
513	185.5	28.5	49.8	2.6	0.1	7.5	7.0
1025	80.1	12.6	76.1	2.8	0.0	3.6	3.1
2049	50.1	8.0	84.1	2.6	0.0	2.3	2.0

Table 5.8: Runtime behavior of skewed two–dimensional blocked red–black Gauss–Seidel.

Grid Size	% of all accesses which are satisfied by				
	±	L1 Cache	L2 Cache	L3 Cache	Memory
2 relaxations combined					
17	3.5	96.5	0.1	0.0	0.0
33	2.9	72.6	24.5	0.0	0.0
65	2.9	60.8	36.1	0.2	0.0
129	2.8	59.2	35.4	2.6	0.0
257	1.6	53.4	40.9	4.1	0.0
513	-0.2	31.6	59.4	8.5	0.7
1025	-3.7	25.5	59.3	17.0	1.9
2049	-3.4	24.0	37.5	38.1	3.9
3 relaxations combined					
17	2.2	97.7	0.1	0.0	0.0
33	1.9	72.7	25.4	0.0	0.0
65	1.9	58.1	39.8	0.2	0.0
129	1.9	56.7	39.4	2.0	0.0
257	1.3	47.4	48.2	3.2	0.0
513	0.3	27.2	65.4	6.6	0.5
1025	-1.6	22.7	59.5	18.2	1.2
2049	-3.0	20.7	30.1	50.2	2.1
4 relaxations combined					
17	0.8	99.2	0.0	0.0	0.0
33	-0.1	71.6	28.4	0.0	0.0
65	-2.8	68.6	34.0	0.2	0.0
129	-0.3	62.5	36.2	1.6	0.0
257	5.7	42.8	48.6	2.9	0.0
513	10.5	28.8	54.9	5.6	0.3
1025	9.6	21.1	46.0	22.4	0.9
2049	8.6	19.0	13.2	57.7	1.4

Table 5.9: Memory access behavior of square two–dimensional blocked red–black Gauss–Seidel.

Grid Size	% of all accesses which are satisfied by				
	±	L1 Cache	L2 Cache	L3 Cache	Memory
2 relaxations combined					
17	39.7	60.2	0.1	0.0	0.0
33	43.6	35.6	20.7	0.1	0.0
65	45.6	35.5	18.2	0.7	0.0
129	46.0	33.6	17.9	2.5	0.0
257	45.9	27.0	24.4	2.7	0.0
513	44.6	20.9	31.1	3.3	0.1
1025	40.9	18.2	28.7	10.4	1.8
2049	39.3	18.4	6.4	34.0	1.8
3 relaxations combined					
17	28.2	71.3	0.3	0.2	0.0
33	40.9	34.8	24.3	0.1	0.0
65	44.7	34.3	20.5	0.5	0.0
129	47.5	31.2	19.3	2.0	0.0
257	48.4	24.8	24.3	2.4	0.0
513	48.0	20.2	26.8	4.9	0.1
1025	46.5	15.8	19.1	17.3	1.2
2049	45.5	16.6	0.2	36.4	1.2
4 relaxations combined					
17	27.1	72.8	0.1	0.0	0.0
33	27.4	43.4	29.1	0.1	0.0
65	33.4	46.3	19.5	0.9	0.0
129	36.9	42.3	19.1	1.7	0.0
257	38.1	34.1	25.1	2.7	0.0
513	38.0	28.3	27.0	6.7	0.1
1025	36.9	24.9	19.7	17.6	0.9
2049	36.2	25.5	0.4	36.9	0.9

Table 5.10: Memory access behavior of skewed two–dimensional blocked red–black Gauss–Seidel.

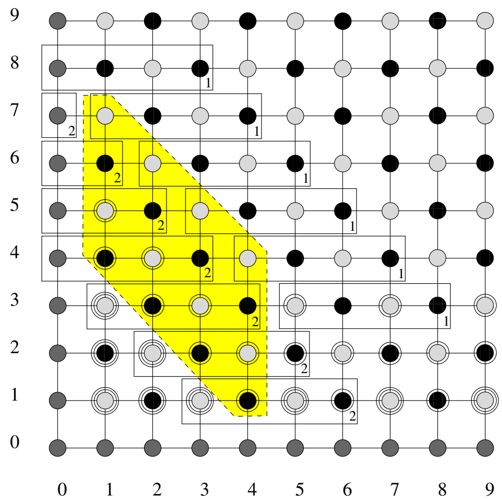

Figure 5.13: Alpha 21164 L1 cache mapping for a 1025×1025 grid.

fraction of the data for the 2049×2049 grid. Consequently, a lage fraction of the data is fetched from the relatively slow L3 cache.

Furthermore, the principle of the square blocked approach prevents the CPU from keeping the directly reused data from the relaxation of a red node and the relaxation of the black node directly below in a register since the red and black nodes are no longer updated in pairs.

In the case of the skewed blocked approach, the utilization of the registers improved slightly, however, the L1 and L2 cache utilization is worsened dramatically especially for the larger grids. Most of the data has to be fetched from the L3 cache. If a 2 Mbyte L3 cache is used even the L3 cache is not able to keep the data and more than 20 per cent of all array references are not cached and therefore lead to memory accesses [WKKR99].

The reason for this is a very high number of cross interference misses in the L1 cache. A visualization with CVT [vdDTGK97] shows that throughout the whole run of the skewed two–dimensional blocked red–black Gauss–Seidel only four cache lines of the direct mapped L1 cache are used simultaneously. Surprisingly, even the L2 cache which is three–way set associative in the case of the Compaq Alpha 21164 processor cannot resolve the conflict misses.

A possible mapping of the nodes within a parallelogram of the skewed approach used for a 1025×1025 grid on the cache lines of the direct mapped Alpha 21164 L1 cache is shown in Figure 5.13. All elements of the uppermost diagonal are mapped to the same cache line (cache line number 1). Hence, the data of the uppermost diagonal is not reused during the update of the second diagonal. Furthermore, the data needed for the update of a single node is producing a conflict in the L1 cache. For example, when updating the red node $(7, 1)$, the nodes $(7, 0)$, $(8, 1)$, $(7, 2)$, $(6, 1)$, and $(7, 1)$ are needed. Thus, accesses to the nodes $(7, 0)$ and $(6, 1)$, $(8, 1)$ and $(7, 2)$, as well as $(8, 1)$ and $(7, 1)$ are causing conflict misses.

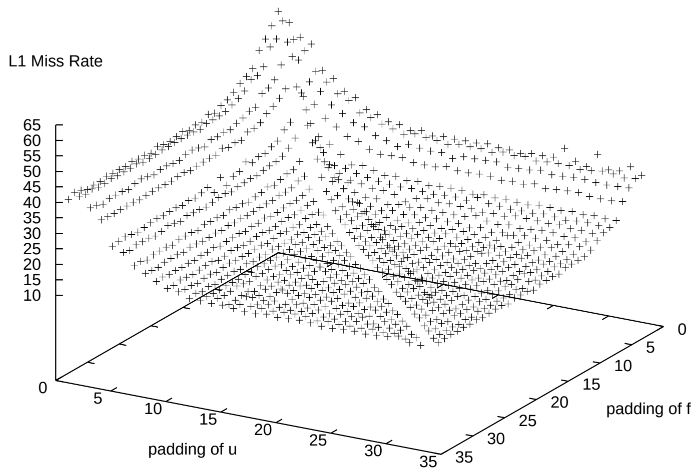

Figure 5.14: L1 miss rate for different padding sizes for 1025×1025.

5.1.5 Array Padding

A common technique to reduce the number of conflict misses is array padding [RT98a, RT98b]. In the case of the red–black Gauss–Seidel method the array padding technique can be applied to two two–dimensional arrays (the unknowns u and the right–hand side f of the equation). In general, array padding can be applied inbetween the two array (*inter array padding*) as well as each of the array dimensions can be padded (*intra array padding*).

The L1 data cache and L2 cache miss rates in per cent for a wide range of intra array padding sizes for the skewed two–dimensional blocking approach applied to a 1025×1025 grid ($B = 4$) are shown in Figures 5.14 and 5.15.

If no padding is applied the miss rates for the L1 cache is extremely high. As soon as a small padding for u or f is used the L1 miss rate drops significantly. Nevertheless, good hit rates can only be achieved if padding for both arrays is applied simultaneously. If equally sized padding for both arrays is used the L1 data cache miss rate is still significantly higher as illustrated by the diagonal in Figure 5.14. The reason for this is cross interference between the two arrays. Cross interference is usually avoided by inter array padding. The study, however, shows that inter array padding is not required for the red–black Gauss–Seidel approach if two differently sized array paddings are used. In contrary to the L1 cache the L2 cache is three–way set associative. Nevertheless, the miss rates are very high if no padding is applied. The overall behavior for various padding sizes, however, is slightly different. For padding sizes which are bad for the L1 cache the L2 cache is accessed more often than for good L1 padding sizes. Since the L2 can resolve many of the conflicts which happen in the L1 cache a relatively high fraction of the accesses hit in

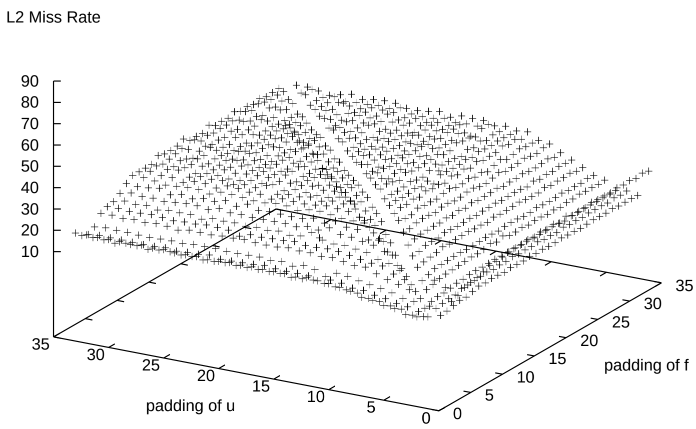

Figure 5.15: L2 miss rate for different padding sizes for 1025×1025.

the L2 cache. For the red–black Gauss–Seidel algorithm this is especially true if the same padding size is used for both arrays.

The runtime performance of the skewed blocked red–black Gauss–Seidel algorithm for a good L1 data cache padding size is summarized in Table 5.11. The performance for all grid sizes except the smallest grid size increases. Especially the performance for the large grid sizes improves by a factor of two to five depending on the grid size and the number of blocked iterations. The skewed two–dimensional blocked algorithm now easily outperforms the one–dimensional blocking technique. The analysis of the runtime spent for execution and different kind of stalls shows that especially the data cache miss stall time is reduced. In contrary to the version without padding, the L1 cache now is efficiently used for all grid sizes (see Table 5.12). Thus, more than 80 per cent of the data is fetched from registers and the L1 cache.

The array padding technique applied to the square blocking technique is able to improve the performance of this approach as well. Figure 5.16 shows the MFLOPS of the square blocked approach using various padding sizes for u and f. In contrary to the skewed approach a relatively small padding is sufficient to avoid severe conflicts. The runtime performance and memory access behavior after padding is summarized in Tables 5.13 and 5.14. The speedup for small grid sizes achieved with padding is not as high as the speedup for the skewed approach. Nevertheless, for the larger grids and especially if many successive red–black Gauss–Seidel iterations are blocked, the technique is able to accelerate the execution of the standard red–black Gauss–Seidel code after array transpose by a remarkable factor of two to four. The performance of the square blocked algorithm after padding is still dominated by data cache misses. In addition the DTB misses, branch miss predictions, and register dependences, however, now have a severe

Grid Size	MFlops	% of cycles used for					
		Exec	Cache	DTB	Branch	Depend	Nops
2 relaxations combined							
17	347.0	59.2	0.4	1.1	3.0	17.9	13.9
33	524.8	66.2	8.9	1.5	0.4	10.6	9.8
65	500.2	65.1	20.3	1.4	0.1	0.3	11.3
129	440.2	54.8	32.6	1.8	0.2	0.2	8.8
257	415.3	53.0	35.1	1.6	0.2	0.2	8.5
513	331.6	41.6	48.9	1.2	0.2	0.2	6.6
1025	190.8	26.4	65.9	0.5	0.0	0.0	6.0
2049	183.3	26.2	65.4	0.5	0.2	0.2	5.5
3 relaxations combined							
17	353.9	41.8	0.9	1.9	3.0	43.0	1.8
33	497.2	49.1	0.6	0.6	2.5	38.6	0.7
65	487.7	44.5	3.0	0.2	0.9	44.2	0.0
129	430.6	51.9	29.9	1.4	0.4	2.7	9.3
257	433.4	51.8	32.7	1.3	0.2	1.6	9.4
513	355.3	42.9	42.5	0.8	0.1	0.6	9.1
1025	237.4	31.3	59.2	0.6	0.0	0.4	6.6
2049	231.6	32.4	56.2	0.6	0.0	0.4	7.2
4 relaxations combined							
17	347.0	43.4	0.4	1.8	3.2	43.9	1.7
33	419.9	49.3	0.9	0.7	5.3	38.8	1.2
65	419.5	47.8	2.6	0.5	1.6	44.1	1.4
129	400.2	50.0	20.8	1.1	0.3	7.3	13.1
257	402.8	49.5	22.8	1.1	0.1	6.3	13.3
513	349.1	44.1	31.6	0.9	0.1	5.0	11.6
1025	267.6	35.3	46.6	0.9	0.1	3.5	10.1
2049	260.7	36.8	45.3	0.5	0.0	3.0	9.1

Table 5.11: Runtime behavior of skewed two–dimensional blocked red–black Gauss–Seidel after array padding.

Grid Size	% of all accesses which are satisfied by				
	±	L1 Cache	L2 Cache	L3 Cache	Memory
2 relaxations combined					
17	36.3	59.5	4.2	0.0	0.0
33	44.7	50.7	4.5	0.1	0.0
65	48.4	43.7	7.3	0.7	0.0
129	47.5	43.0	6.9	2.6	0.0
257	45.2	44.6	7.6	2.6	0.0
513	44.1	45.1	7.2	3.1	0.5
1025	40.7	46.2	8.2	3.0	1.8
2049	39.2	48.3	7.3	3.5	1.8
3 relaxations combined					
17	28.9	69.8	1.2	0.0	0.0
33	39.5	55.3	5.1	0.1	0.0
65	43.6	45.8	10.0	0.6	0.0
129	46.7	41.0	10.3	2.0	0.0
257	48.0	40.3	9.5	2.2	0.0
513	46.8	41.4	8.8	2.6	0.3
1025	46.2	42.6	7.7	2.3	1.2
2049	45.3	43.1	7.9	2.4	1.2
4 relaxations combined					
17	26.1	72.7	1.2	0.0	0.0
33	28.2	66.4	5.3	0.1	0.0
65	34.3	55.7	9.1	0.9	0.0
129	37.5	51.7	9.0	1.9	0.0
257	37.8	52.8	7.0	2.3	0.0
513	38.4	52.7	6.2	2.4	0.3
1025	36.7	54.3	6.1	2.0	0.9
2049	35.9	55.2	6.0	1.9	0.9

Table 5.12: Memory access behavior of skewed two–dimensional blocked red–black Gauss–Seidel after array padding.

Grid Size	MFlops	% of cycles used for					
		Exec	Cache	DTB	Branch	Depend	Nops
2 relaxations combined							
16	107.2	45.0	0.1	3.2	5.7	39.1	2.0
32	110.5	45.3	0.3	3.2	4.5	40.8	2.2
64	110.2	43.4	1.6	3.5	4.0	41.6	2.2
128	104.8	42.4	5.6	3.4	3.8	40.1	2.1
256	99.2	39.7	12.0	5.4	3.8	33.6	2.0
512	85.0	35.6	18.5	7.8	3.0	30.3	0.4
1024	72.9	29.2	26.5	12.5	2.4	25.1	1.1
2048	65.2	27.6	26.0	15.5	4.5	22.4	1.7
3 relaxations combined							
16	114.9	41.8	0.1	1.7	7.2	42.3	2.2
32	119.6	41.6	0.4	1.6	6.5	44.1	2.4
64	120.4	40.2	1.1	1.8	5.5	45.1	2.4
128	115.2	38.5	4.3	1.7	4.9	43.8	2.3
256	114.6	38.5	8.8	3.1	4.9	39.4	2.3
512	101.5	35.3	15.8	4.1	3.8	35.7	0.4
1024	90.1	30.0	23.9	5.9	4.0	30.4	1.2
2048	85.8	30.7	22.4	8.7	4.6	29.3	2.0
4 relaxations combined							
16	190.3	55.2	0.3	1.6	5.2	26.4	5.5
32	237.7	63.9	1.3	2.3	3.7	16.0	6.8
64	255.0	69.2	3.2	3.4	2.4	8.3	8.2
128	220.1	60.7	13.5	8.1	1.6	3.7	7.3
256	225.3	59.0	20.6	2.7	0.9	1.7	7.5
512	228.7	60.6	22.7	3.3	1.1	0.9	6.0
1024	182.8	51.3	30.1	6.2	0.7	0.4	6.2
2048	167.6	49.4	28.0	10.7	0.7	0.5	6.8

Table 5.13: Runtime behavior of square two–dimensional blocked red–black Gauss–Seidel after array padding.

Grid Size	% of all accesses which are satisfied by				
	±	L1 Cache	L2 Cache	L3 Cache	Memory
2 relaxations combined					
17	3.0	96.8	0.2	0.0	0.0
33	3.0	89.2	7.8	0.0	0.0
65	2.7	85.4	11.4	0.4	0.0
129	2.6	86.8	7.3	3.3	0.0
257	1.4	86.7	4.9	7.0	0.0
513	-0.3	88.2	5.8	5.6	0.7
1025	-3.4	90.6	3.9	6.9	1.9
2049	-4.5	91.0	4.4	7.2	1.9
3 relaxations combined					
17	2.1	97.7	0.2	0.0	0.0
33	2.0	92.8	5.1	0.0	0.0
65	2.0	87.4	10.3	0.3	0.0
129	1.4	87.0	9.2	2.4	0.0
257	1.3	88.7	5.6	4.4	0.0
513	0.6	89.5	5.5	4.2	0.1
1025	-1.4	91.4	4.7	4.1	1.3
2049	-3.0	92.3	5.0	4.0	1.7
4 relaxations combined					
17	1.0	98.6	0.3	0.0	0.0
33	-0.5	94.8	5.7	0.0	0.0
65	-2.1	90.6	11.2	0.3	0.0
129	-1.7	82.1	17.5	2.0	0.0
257	5.9	75.2	14.7	4.1	0.0
513	12.6	77.6	6.6	3.1	0.1
1025	9.3	81.2	5.9	2.7	1.0
2049	8.7	81.4	6.1	2.7	1.1

Table 5.14: Memory access behavior of square two–dimensional blocked red–black Gauss–Seidel after array padding.

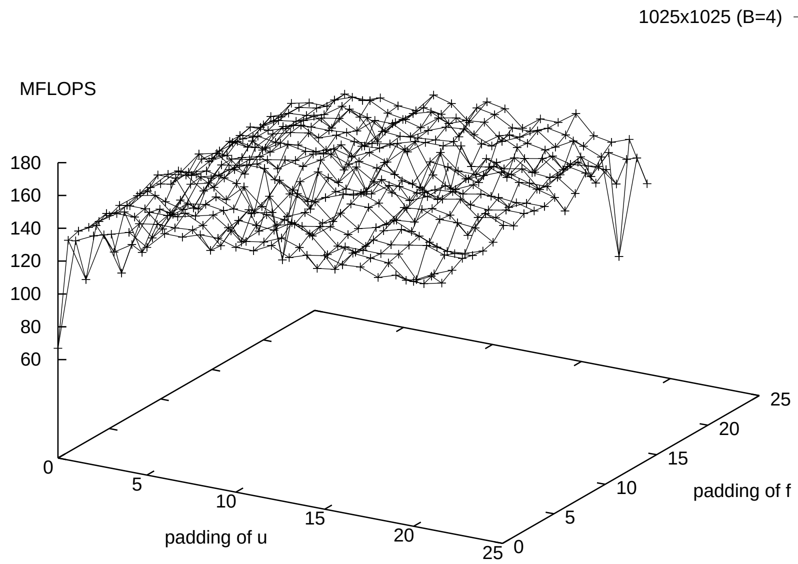

Figure 5.16: MFLOPS for square two–dimensional blocked red–black Gauss–Seidel for various padding sizes.

impact on the performance as well.

The presented experiments use an exhaustive search to find favorable padding sizes. In general, however, this is not applicable in a compiler environment. The heuristics developed by Tseng et al. [RT98a], for example, can be used to calculate padding sizes within a compiler. Their *IntraPad* heuristics proved to find reasonable paddings for the red–black Gauss–Seidel algorithm.

5.1.6 Summary of Performance Results

The runtime performance of all red–black Gauss–Seidel codes after applying the optimization techniques on a Compaq PWS 500au is summarized in Figure 5.17. Although the array transpose technique is one of the simplest techniques it increases the performance for all grid sizes. However, the performance for the larger grids which don't fit in the L3 cache of the Compaq PWS 500au is still only about 60 MFLOPS. The fusion and blocking techniques with three blocked iterations can further improve the performance for all grid sizes. For the small grids, which fit in the L2 cache, the additional improvement is only marginal. Nevertheless, the performance of 560 MFLOPS for a 65×65 grid is very impressive. The greatest speedup is achieved for the grids which do no longer fit in the L2 cache. For the 257×257 grid the one–dimensional blocking technique achieves a speedup of 3.8 compared to the standard red–black Gauss–Seidel version and a speedup

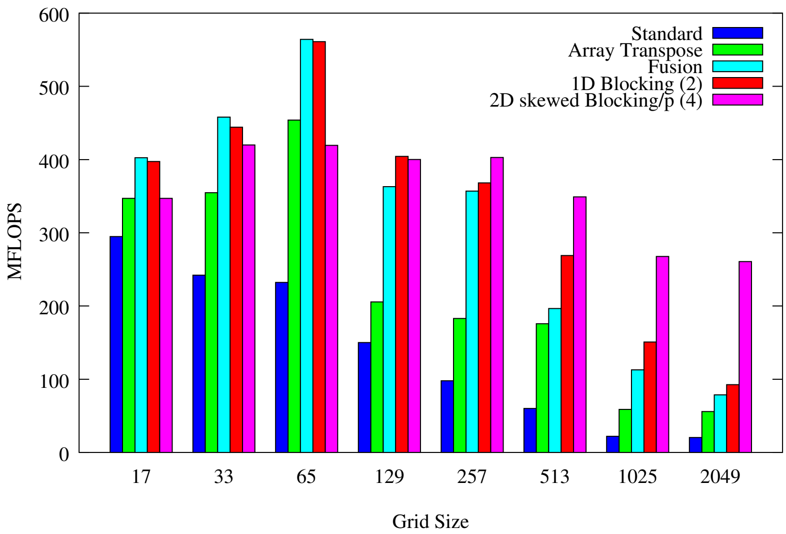

Figure 5.17: MFLOPS for different red–black Gauss–Seidel variants on a Compaq PWS 500au.

of 2.0 compared to the version after array transpose. Although the blocking technique is able to increase the performance of the algorithm for the very large grids slightly, the performance is still far away from the achievable peak performance of one GFLOPS. The skewed and square two–dimensional blocking techniques without array padding are not performing satisfactorily. For the small grids and for the larger grids, the floating point performance is equal or even worse than the performance of the standard implementation after array transpose. The padded version of the skewed approach, however, achieves a remarkable speedup especially for the large grids. Therefore, only the performance of the skewed two–dimensional blocking technique after array padding performing four update sweeps simultaneously is shown in Figure 5.17. For the larger grids the two–dimensional blocking technique is able to sustain a performance of 260 MFLOPS. Compared to the standard red–black Gauss–Seidel code this is a speedup of 12.8 and a speedup of 4.7 compared to the code after array transpose. Note, that an improved MFLOPS rate is equivalent to a similar speedup in runtime since the number of executed floating point operations is not changed by the optimiaztion techniques.

In summary, the programmer should make sure that an appropriated data layout is used. Thus, *array transpose* and *array padding* is recommended in the early optimization phase. The *fusion technique* is useful when the data set size is relatively small, i.e. the amount of cache misses is relatively small. The fusion technique will improve hit rates with relatively low loop overhead. With growing grid size, the *one–* resp. *two–dimensional blocking* technique should be used. The one–dimensional blocking technique is easier to apply and implies less loop overhead than the two–dimensional blocking technique. Furthermore, the technique is less vulnerable to self interference. The two–dimensional blocking technique, however, should be used especially for very large grids

Grid Size	Pentium 3 960 MHz			Pentium 4 1.5 GHz		
	MFlops		Speedup	MFlops		Speedup
17	165.7	201.3	1.2	142.8	205.3	1.4
33	175.4	222.1	1.3	134.6	202.5	1.5
65	396.5	447.8	1.1	331.8	628.2	1.9
129	104.9	385.0	3.7	327.1	565.6	1.7
257	73.3	302.1	4.1	294.6	542.9	1.9
513	71.2	291.4	4.1	312.4	518.2	1.7
1025	72.3	231.7	3.2	314.5	493.4	1.6
2049	70.6	216.3	3.1	307.5	437.8	1.4
	Athlon (G75) 700 MHz			Athlon (Tbird) 1.2 GHz		
	MFlops		Speedup	MFlops		Speedup
17	251.0	383.0	1.5	401.2	627.5	1.6
33	261.3	431.4	1.7	425.5	692.1	1.6
65	474.7	504.0	1.1	774.1	802.8	1.0
129	333.7	499.3	1.5	434.6	739.2	1.7
257	111.5	396.5	3.6	103.1	490.8	4.8
513	100.8	325.1	3.2	101.1	483.4	4.8
1025	100.3	270.1	2.7	101.2	460.8	4.6
2049	92.3	231.5	2.6	91.5	271.2	3.0

Table 5.15: 5–point stencil red–black Gauss–Seidel performance on PCs with a 960 MHz Intel Pentium 3, 1.5 GHz Intel Pentium 4, 700 MHz AMD Athlon (G75 core), resp. a 1.2 GHz AMD Athlon (Thunderbird core). The left MFLOPS column in each block illustrates the standard red–black Gauss–Seidel performance whereas the right column shows the highest achieved performance after applying the locality optimizations.

where a small amount of grid lines does no longer fit in the higher levels of the cache to improve L1 and L2 cache hit rates. To ensure good hit rates, however, additional array padding may be necessary

The applicability of the techniques described in this thesis is not limited to the architecture of the Compaq PWS 500au. Remarkable speedups can also be obtained on other workstations and PCs based on x86 compatible microprocessors. Table 5.16 and Table 5.15 compare MFLOPS for a red–black Gauss–Seidel implementations after array transpose with the best implementation obtained by applying data locality optimizations. The red–black Gauss–Seidel implementations were executed on a Compaq PWS 500au (based on a 500 MHz Alpha 21164), a Compaq XP1000 (based on a 500 MHz Alpha 21264), a SUN Ultra60 (based on a 295 MHz UltraSparcII), an SGI Origin 2000 node (based on a 195 MHz R10000), a HP SPP2200 Convex Exampler node (based on a 200 MHz PA–8200), a HP N–Class node (based on a 440 MHz PA–8500), and Linux PCs (based on a 960 MHz Intel Pentium 3 Coppermine, 1.5 GHz Intel Pentium 4, 700 MHz

Grid Size	Alpha 21164			Alpha 21264		
	MFlops		Speedup	MFlops		Speedup
17	347.0	402.6	1.2	491.5	589.8	1.2
33	354.8	458.0	1.3	629.8	699.8	1.1
65	453.9	564.2	1.2	650.3	812.9	1.3
129	205.5	404.3	2.0	330.3	717.7	2.2
257	182.9	402.8	2.2	332.9	755.2	2.3
513	175.9	349.1	2.0	196.6	710.6	3.6
1025	58.8	267.6	4.6	119.6	415.9	3.5
2049	55.9	260.7	4.7	109.9	396.4	3.6
	SUN UltraSparc II			SGI R10000		
	MFlops		Speedup	MFlops		Speedup
17	132.2	173.2	1.3	187.0	252.3	1.3
33	152.2	185.2	1.2	207.2	287.3	1.4
65	99.3	169.7	1.7	147.0	260.6	1.8
129	102.4	154.1	1.5	151.9	261.6	1.7
257	102.1	143.2	1.4	173.6	250.3	1.4
513	51.6	113.6	2.2	141.6	195.3	1.4
1025	45.6	113.4	2.5	66.9	163.2	2.4
2049	42.2	112.7	2.7	66.8	150.5	2.3
	HP PA–8200			HP PA–8500		
	MFlops		Speedup	MFlops		Speedup
17	210.7	280.9	1.3	491.5	737.3	1.5
33	286.3	349.9	1.2	629.8	787.3	1.3
65	361.3	427.8	1.2	812.9	1083.8	1.3
129	412.9	480.2	1.2	1101.1	1101.1	1.0
257	416.2	511.2	1.2	665.9	1107.6	1.7
513	52.2	245.7	4.7	107.8	552.7	5.1
1025	49.3	231.7	4.7	108.0	540.7	5.0
2049	43.8	225.6	5.1	104.8	543.2	5.2

Table 5.16: 5–point stencil red–black Gauss–Seidel performance on a Compaq PWS 500au (500 MHz A21164), a Compaq XP1000 (500 MHz A21264), a SUN Ultra60 (295 MHz UltraSparcII), an SGI Origin 2000 node (195 MHz R10000), a HP SPP2200 Convex Exampler node (200 MHz PA–8200), and a HP N–Class node (440 MHz PA–8500). The left MFLOPS column in each block illustrates the standard red–black Gauss–Seidel performance whereas the right column shows the highest achieved performance after applying the locality optimizations.

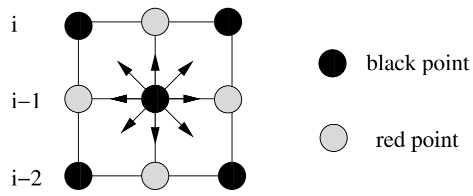

Figure 5.18: Data dependences in a 9–point stencil red–black Gauss–Seidel algorithm.

AMD Athlon (G75 core), and a 1.2 GHz AMD Athlon (Thunderbird core)).

The Compaq XP1000 delivers a peak performance of one GFLOPS as the Compaq PWS 500au does. However, besides the addition of architectural improvements such as out–of–order execution ability the computer manufacturer significantly improved the memory throughput of the machine. Consequently, the improvement achieved by the data locality optimizations is smaller for the newer machine. Both HP machines are equipped with a single level of a large cache. In the case of the PA–8200 the cache is located off chip whereas the PA–8500 is equipped with a large on–chip cache. That design implicates that the cache optimization techniques for the small grid sizes have no significant impact on the performance of the red–black Gauss–Seidel code. As soon as the data no longer fits in the cache, however, the performance of the standard code (after array transpose) drops dramatically. For the same grid sizes the optimized codes are able to sustain 220 MFLOPS on a HP SPP2200 Convex Exampler node (PA–8200) and remarkable 540 MFLOPS on a HP N–Class node (PA–8500).

The performance of the red–black Gauss–Seidel codes on the Linux PCs is competitive with that of workstations. Thereby, two interesting facts arise. First, the Pentium 4 system is able to deliver a standard performance for the larger grids which is close to or even higher than the optimized performance on several workstations. The reason for this is that the Pentium 4 system uses RDRAM which allows a higher main memory bandwidth and lower latency than older memory system such as built in the three other Linux PCs. The performance optimizations are able to further improve the performance, although the improvement is much smaller then the improvement achieved for the workstations or other PCs. Second, both Athlon based systems achieve more or less the same MFLOPS rates for the standard red–black Gauss–Seidel code (after array transpose) although there is a big difference in peak performance (1.4 GFLOPS to 2.4 GFLOPS). Both machines are equipped with the same memory system. Thus, the fact that the performance for the grids larger then 129×129 is similar on both machines proves that the runtime is determined by the main memory speed.

5.2 Nine Point Stencil Discretization

All techniques described above were explained while assuming a 5–point stencil discretization. Using a 9–point stencil discretization (see Figure 5.18) four additional data

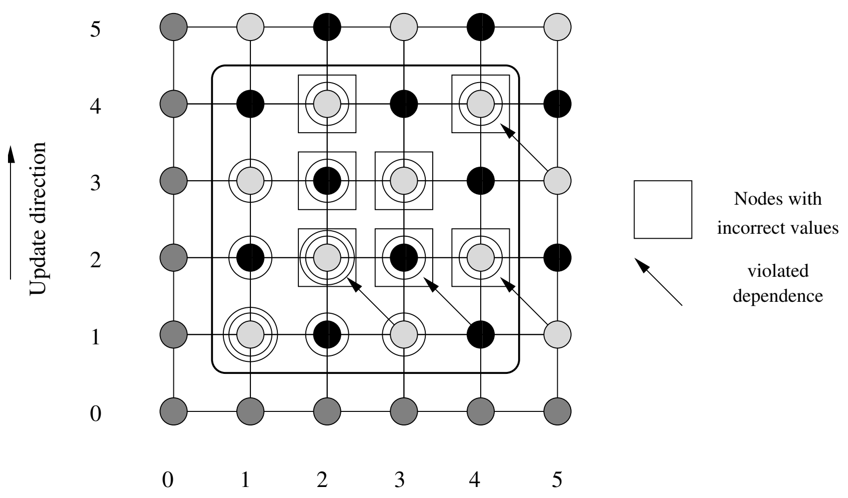

Figure 5.19: Square two–dimensional blocking violates data dependences in the 9–point stencil case.

dependences to the neighboring nodes in the north west, north east, south west, and south east must be obeyed. Red points resp. black points are not independent of each other anymore and have to be updated in certain order. Since the optimization techniques described in this thesis produce bitwise identical results these additional data dependences must be obeyed as well.

The optimization techniques *fusion* and *one–dimensional blocking* assume that if the grid is updated from bottom to top a node in row $i-1$ can be updated as soon as the node directly above it in line i is up to date. This assumption is sufficient to obey the additional data dependences of the 9–point stencil discretization. Thus, the *fusion* and *one–dimensional blocking* technique can be applied to a red–black Gauss–Seidel smoother based on 9–point discretization without any modifications.

However, matters are more complicated for the two–dimensional blocking techniques. If the square two–dimensional blocking technique is used to relax a grid as shown in Figure 5.19 the update of the first tile in the lower left corner already violates data dependences. The first update of the red node $(2, 4)$ is not allowed since the red node $(1, 5)$ hasn't been relaxed so far. If the dependence is ignored by the square two–dimensional blocking technique the value of the node $(2, 4)$ will be incorrect after relaxation and consequently the values of other nodes will be incorrect as well. If the algorithm is modified so that only nodes are updated which won't violate a data dependence the nodes are updated on diagonals in a fashion similar to the skewed two–dimensional blocking approach.

The data dependences within the parallelogram used by the skewed two–dimensional blocking technique combining two relaxations ($B = 2$) for a 5–point stencil discretization are shown in Figure 5.20 on the left side. Each arrow represents a dependence. For example, node number 4 must be updated before node number 7 or 8 can be updated. Whereas node number 3 and 4 can be updated in an arbitrary order. Hence, there are many

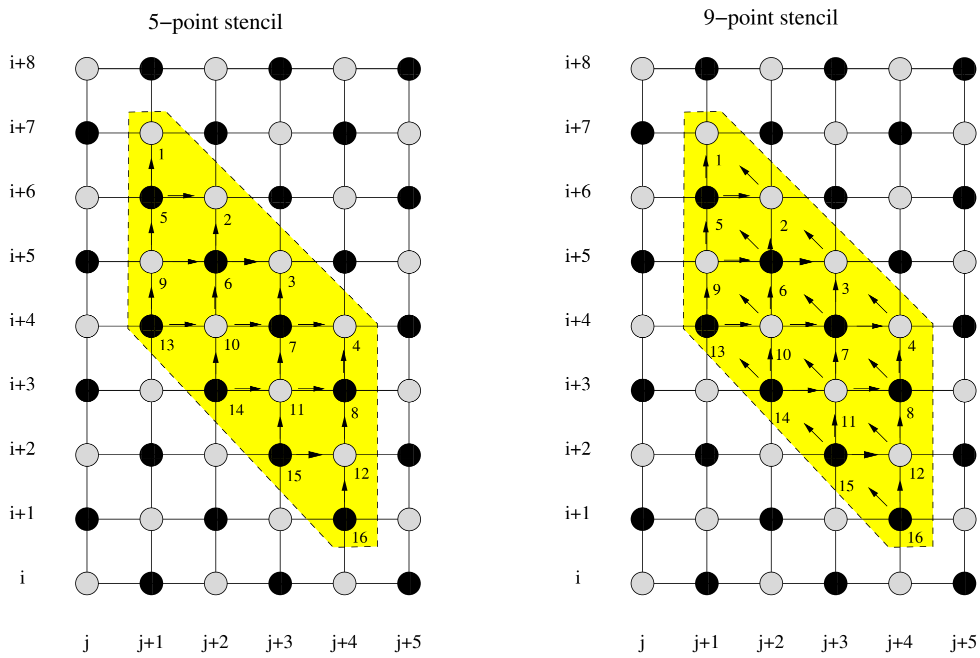

Figure 5.20: 5–point stencil and 9–point stencil data dependences within the skewed block.

possible update orders. E.g. updating all red nodes in arbitrary order in the uppermost diagonal and then updating all black nodes in the diagonal below and so on. If a 9–point stencil discretization is used (Figure 5.20 on the right side) the programmer must take care of an additional data dependence of a node to its neighboring node in the north east. For example, node 1 can be relaxed as soon as node 2 has been relaxed. Hence, the nodes on a diagonal can only be updated from bottom to top.

The performance of a standard red–black Gauss–Seidel implementation based on a 9–point discretization of the differential operator after array transpose compared to the best possible performance obtained by the previously described optimizations on several workstations and Linux PCs is shown in Table 5.17 and Table 5.18[4]. For most workstations the standard performance for the 9–point stencil code is higher than that for the 5–point stencil codes. Only on the Linux based machines the 9–point stencil case performs significantly worse. The data locality optimization techniques are able to speed up the computation on all machines. The achieved floating point performance after applying the optimizations is more or less equal to the performance in the 5–point stencil case. Consequently, the performance gain in the 9–point stencil case is not as high as in the 5–point stencil case.

[4]The 9–point stencil benchmark results for the HP PA–8200 and HP PA–8500 have been omitted since the results are not representive due to compiler instabilities during the optimization phase of the program compilation.

Grid Size	Alpha A21164			Alpha 21264		
	MFlops		Speedup	MFlops		Speedup
17	408.34	408.34	1.0	530.8	530.8	1.0
33	472.35	515.29	1.1	629.8	629.8	1.0
65	585.25	650.28	1.1	688.5	731.6	1.1
129	330.32	540.26	1.6	457.4	724.7	1.6
257	323.93	534.02	1.6	461.0	747.6	1.6
513	113.51	335.35	3.0	273.5	663.3	2.4
1025	103.04	266.96	2.6	188.4	463.7	2.5
2049	90.06	239.99	2.7	177.5	388.2	2.2
	SUN UltraSparc II			SGI R10000		
	MFlops		Speedup	MFlops		Speedup
17	120.65	163.34	1.4	217.56	279.57	1.3
33	138.25	175.30	1.3	230.41	292.47	1.3
65	109.39	165.62	1.5	198.06	290.56	1.5
129	112.19	145.84	1.3	221.03	289.47	1.3
257	112.01	133.17	1.2	195.52	282.35	1.4
513	69.55	119.68	1.7	162.16	251.87	1.6
1025	55.30	119.79	2.1	98.10	221.03	2.3
2049	49.66	118.64	2.4	90.74	216.73	2.4

Table 5.17: 9–point stencil red–black Gauss–Seidel performance on a Compaq PWS 500au (500 MHz A21164), a Compaq XP1000 (500 MHz A21264), a SUN Ultra60 (295 MHz UltraSparcII), an SGI Origin 2000 node (195 MHz R10000), a HP SPP2200 Convex Exampler node (200 MHz PA–8200), and a HP N–Class node (440 MHz PA–8500). The left MFLOPS column in each block illustrates the standard red–black Gauss–Seidel performance whereas the right column shows the highest achieved performance after applying the locality optimizations.

Grid Size	Pentium 3 960 MHz			Pentium 4 1.5 GHz		
	MFlops		Speedup	MFlops		Speedup
17	98.5	112.3	1.1	99.3	162.0	1.6
33	102.6	117.6	1.1	98.7	159.8	1.6
65	219.0	275.5	1.3	406.4	427.8	1.1
129	99.0	203.9	2.1	390.9	410.3	1.0
257	63.8	191.8	3.0	354.2	396.3	1.1
513	71.7	190.6	2.7	363.3	397.9	1.1
1025	70.5	159.6	2.3	362.0	394.0	1.1
2049	68.2	121.1	1.8	228.6	370.4	1.6
	Athlon (G75) 700 MHz			Athlon (Tbird) 1.2 GHz		
	MFlops		Speedup	MFlops		Speedup
17	186.1	209.9	1.1	299.4	335.1	1.1
33	195.0	254.0	1.3	310.2	409.0	1.3
65	285.2	293.4	1.0	454.7	478.1	1.1
129	233.4	281.0	1.2	207.8	402.6	1.9
257	78.4	242.5	3.1	75.8	331.9	4.4
513	69.1	212.8	3.1	74.9	305.6	4.1
1025	69.0	189.6	2.7	74.3	292.5	3.9
2049	61.5	172.7	2.8	73.2	201.5	2.8

Table 5.18: 9–point stencil red–black Gauss–Seidel performance on PCs with a 960 MHz Intel Pentium 3, 1.5 GHz Intel Pentium 4, 700 MHz AMD Athlon (G75 core), resp. a 1.2 GHz AMD Athlon (Thunderbird core). The left MFLOPS column in each block illustrates the standard red–black Gauss–Seidel performance whereas the right column shows the highest achieved performance after applying the locality optimizations.

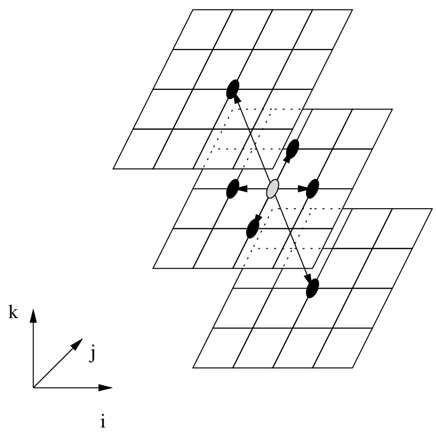

Figure 5.21: 7–point stencil discretization

5.3 Optimizations for Three–dimensional Methods

Physical phenomena simulated with multigrid methods are often of three–dimensional nature and a two–dimensional mapping of the problem is not always useful. Solving three–dimensional problems with similar resolution requires much bigger data structures. A two–dimensional problem represented by a 1025×1025 grid will consume 8 Mbyte of memory for the solution vector of the system of linear equations, for example. A three–dimensional problem represented by a grid with similar resolution already will consume 8 Gbyte of storage space. Furthermore, more storage space will be required for the matrix coefficients, the right–hand side of the equation, and coarser grids in the multigrid context. Thus, the storage requirements by far exceed the size of caches in modern microprocessors even for relatively small three–dimensional grids.

The reuse of data in the cache is further complicated by the fact that the accesses to the same data are usually too far apart in time. Consequently, data elements have to be read from main memory several times during one sweep through the grid. The problem is more grave in the 3D case than in the 2D case as will be explained with the following 3D example. Consider the 7–point stencil placed over a node of a three dimensional grid in Figure 5.21. For the relaxation of the node in the center of the stencil data from three different planes of the grid is required. Assume that the relaxation is performed along one horizontal plane in the grid before nodes in other planes are relaxed.

The situation within the middle plane illustrated in Figure 5.21 is similar to the situation in the two–dimensional case. A 5–point stencil is moved through the plane and as explained in Section 3.3 some data locality will be exploited even if no cache optimizations are applied. The data within the upper plain, however, is reused as soon as the algorithm finished updating all nodes within the middle plain. Thus, the data will only be cached if the whole data of three planes fit in the cache simultaneously. For reasonably large grids this requirement will not be fulfilled. Hence, the data of the upper plain will not be reused and have to be reloaded from main memory once the algorithm relaxes the

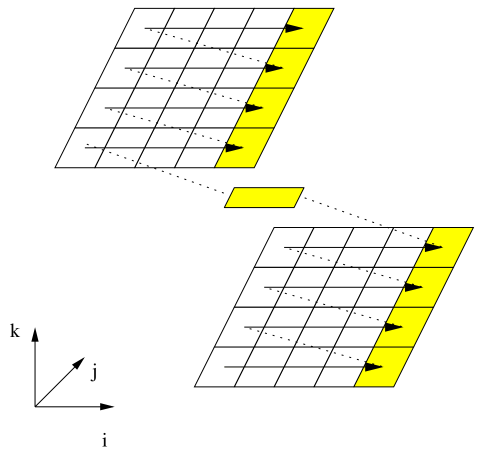

Figure 5.22: Array padding for three–dimensional grids.

nodes within that plain. Similarly, the data will be reloaded a third time once the nodes within the plain directly above the upper plain are relaxed. Furthermore, the data of different plains is widely spread over memory and can cause conflicts in the data translation look–aside buffer. Interchanging the loop order or transposing the data structure will not ease the situation since the new order will cause conflicts in vertical plains.

The solution for both problems are loop blocking techniques. Similar to the two–dimensional case the data dependences have to be carefully observed. The square two–dimensional blocking approach can be adopted to the three–dimensional case as follows.

A relatively small cuboid of nodes is moved through the original grid, starting in its front bottom left corner. First, all red nodes inside the current position of the cuboid have to be relaxed. Then, it is re–positioned within the original grid so that it is moved one step towards the left, one step towards the front, and one step towards the bottom boundary plane of the grid. Of course, the cuboid cannot exceed the boundary planes of the grid. Therefore, a special handling of grid planes located close to the boundaries of the original grid structure needs to be implemented. After re–positioning the cuboid, the black points inside its new position can be updated, before the cuboid is again moved on to its next position. If several red–black Gauss–Seidel iterations are to be blocked, the next position is obtained by moving the cuboid once again one plain in each space dimension. Finally, as soon as all red–black Gauss–Seidel iterations have be processed with the cuboid, the subsequent position of the cuboid is adjacent to the position it had before the re–positioning.

However, this technique by itself will not lead to significant speedups on most modern microprocessors. Similar to the two–dimensional case, interfering data within the blocks can produce a high amount of conflict misses which easily overcome the performance improvement by the tiling. For strategies how to automatically choose good tile and padding sizes for three–dimensional multigrid codes the reader is refered to the work done be Tseng et al. [RT00] which follows the work presented in this thesis.

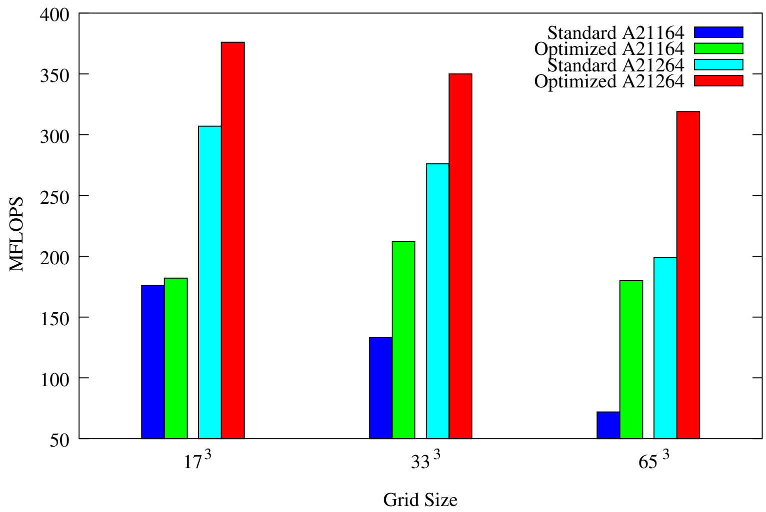

Figure 5.23: MFLOPS for a 3D red–black Gauss–Seidel algorithm on a Compaq PWS 500au and a Compaq XP1000.

A possible array padding technique in the three–dimensional case is illustrated in Figure 5.22. First, padding in i–direction is introduced in order to avoid cache conflict misses caused by grid points which are adjacent in dimension j. Note, that this is a similar padding as in the two–dimensional case. Second, another padding is added between neighboring planes of the grid. This kind of padding is illustrated by the box between the two planes in Figure 5.22. The purpose of the second padding is to avoid conflicting nodes adjacent in z direction.

A standard array padding will add an inter plane padding by extending the j dimension of the array. This will add additional rows to each plane. The size of the inter plane padding is then equal to a multiple of the size of a grid row. In the multigrid context the size of the new padding will likely be a multiple of a large power of two, so that the introduced padding will avoid non or at least not all conflicts between planes. Therefore, a non–standard array padding is required which adds a padding with a size independent of the size of a grid row. Unfortunately, most languages will not support this kind of array index calculation, so that compiler support or a hand coded array linearization is required.

As an example the speedups for a three–dimensional red–black Gauss–Seidel code achieved by applying the blocking and padding techniques described above on a Compaq PWS 500au and Compaq XP1000 are shown in Figure 5.23.

5.4 Problems with Variable Coefficients

This section summarizes the work done by Pfänder [Pfä00] and Kowarschik [Kow01]. Their research is based on the optimization techniques described in this thesis and extends them to variable coefficient problems. The focus of their work is on appropriate data

u_0	ce_0	we_0	ea_0	no_0	so_0	f_0	...	u_n	ce_n	we_n	ea_n	no_n	so_n	f_n

Figure 5.24: Equation–oriented storage scheme.

layout strategies to avoid conflict misses and to maximize spatial locality.

The main difference of constant and variable coefficient problems can be seen by means of the equation of a single grid node. If a 5–point discretization of the differential operator is used the equation for a single inner grid point is:

$$so_i * u_{so_i} + we_i * u_{we_i} + ce_i * u_i + ea_i * u_{ea_i} + no_i * u_{no_i} = f_i. \tag{5.1}$$

In the case of constant coefficient problems each coefficient so_i, we_i, ce_i, ea_i, resp. no_i will have a single value for all grid points. Thus, an iterative method needs to store only five constant values besides the unknown vector u and the right–hand side of the equation f.

For variable coefficient problems five values must be stored for each grid point. Hence, the storage space required for the coefficients outnumbers the storage requirement for u and f. Kowarschik and Pfänder introduced several possible data layouts for u, f and the coefficients. Among them:

- Equation–oriented storage scheme:

  The data value of a grid point and the appendant coefficients of the equation are stored adjacently as shown in Figure 5.24.

- Band–wise storage scheme:

  The unknowns u and the right–hand side of the equation f are stored in separate arrays in memory. Furthermore, the coefficients for each orientation are stored in a separate array. The data layout is illustrated in Figure 5.25.

- Access–oriented storage scheme:

  The vector u is stored in a separate array. Then, for each grid point the right–hand side f and the five corresponding coefficients are stored adjacently as illustrated in Figure 5.26.

The experiments of Kowarschik and Pfänder showed that the equation–oriented storage scheme does not yield good performance for multigrid methods. The reason for this is an inherent inefficiency of the data layout. Whenever a node i is relaxed the current approximations of its four neighboring nodes are required. The equation–oriented storage scheme stores the value of a node, the right–hand side, and the coefficients of that node

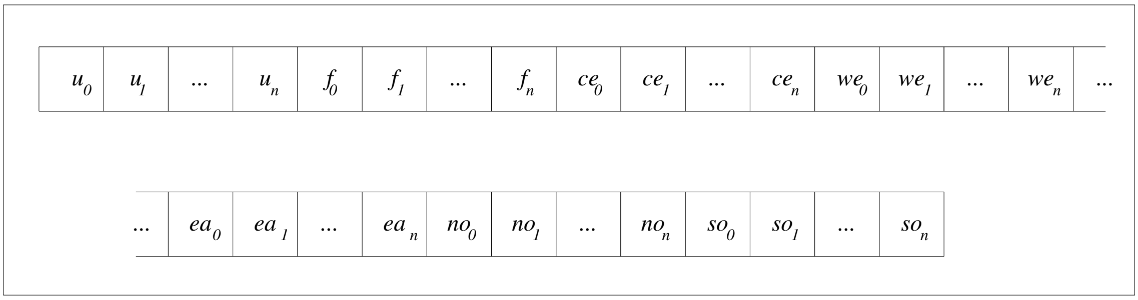

Figure 5.25: Band–wise storage scheme.

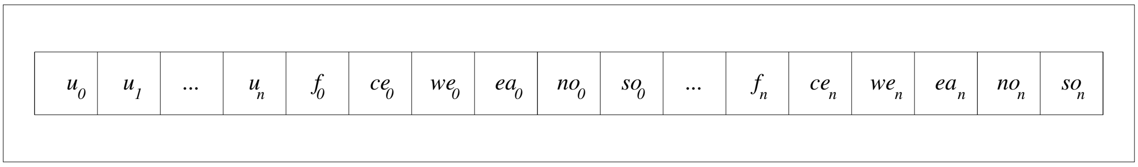

Figure 5.26: Access–oriented storage scheme.

adjacent to each other. Thus, if the value of a node is loaded these values will be loaded into cache as well. Since for the update of the node i these values are not immediately needed bandwidth and cache storage is wasted.

Better performance can be achieved with the band–wise and access–oriented storage scheme. It turned out, however, that the band–wise storage scheme is very vulnerable to array interference. Hence, on many architectures padding of the arrays is indispensable. Contrary, array padding does not affect the performance of the access oriented scheme since the access–oriented storage scheme can be derived from the band–wise storage scheme when the arrays for $f, so_i, ea_i, we_i,$ and no_i are merged. Thus, conflict misses are already avoided by the array merging technique. The access–oriented data layout agglomerates the right–hand side of the equation for a node and its appendant coefficients in memory. Thus, all values required for the update of a node except the unknowns are stored close together and will jointly be loaded into cache.

Execution times of the band–wise and access–oriented storage scheme used in a multigrid code written in C are shown in Figure 5.27. The code executed ten V(2,2) cycles starting on a 1025×1025 grid[5].

Kowarschik and Pfänder also demonstrated that the *fusion*, *one–dimensional*, and *two–dimensional blocking* (square blocked approach) developped in this thesis can be applied to variable coefficient multigrid codes. In the case of variable coefficients a performance improvement by a factor of two to three was achieved.

[5] The experiment was performed on a Compaq XP1000 (Alpha 21264, 500 MHz, Compaq cc V5.9), Compaq PWS 500au (Alpha 21164, 500 MHz, Compaq cc V5.6), an AMD Athlon based PC (700 MHz, gcc), an Intel Pentium 2 based PC (350 MHz, gcc), and an Intel Celeron based PC (400 MHz, gcc).

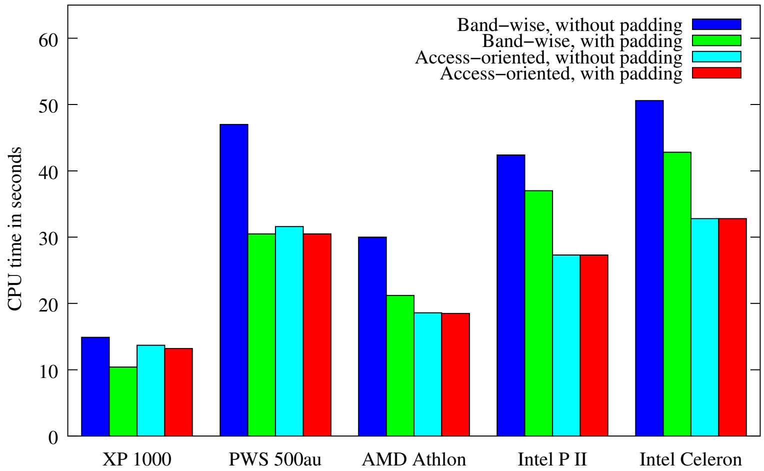

Figure 5.27: CPU times for the multigrid codes based on different data layouts with and without array padding.

5.5 Summary

Standard data locality optimizations cannot directly be applied to stencil based iterative methods since they would violate data dependences. In this chapter, new data locality optimizations have been propsed by means of the red–black Gauss–Seidel method which can be applied to improve the performance of iterative methods. The new data locality optimizations include the data access transformations *fusion*, *one–*, and *two–dimensional blocking*. Furthermore, the application and impact of data layout transformations like *array transpose*, *array padding*, and *array merging* has been demonstrated.

The effect of the optimizations is pointed out with detailed runtime and memory access behavior analysis on a Compaq PWS 500au. Furthermore, the data locality properties were examined with a focus on aspects involved with the multilevel nature of the memory hierarchy.

The performance of a two–dimensional red–black Gauss–Seidel code for constant coefficient problems based on a 5–point discretization of the differential operator can be improved for all grid sizes. For the small grid sizes a performance of about 40 per cent of the peak performance of a Compaq PWS 500au can be achieved. For larger grids which no longer fit in any data cache a performance of about 25 per cent of the peak performance of a Compaq PWS 500au can be achieved. This equals a speedup of 12.8 compared to a straightforward red–black Gauss–Seidel implementation and a speedup of 4.7 compared to a code after array transpose. Comparable speedups have been demonstrated on other machines as well.

The performance improvement is mainly due to data access transformations. However, the two–dimensional blocking techniques tend to suffer from cache–interference phenomena. The impact of the phenomena is especially crucial for multigrid methods which use standard grid coarsening, since they usually use grid sizes which are multiples of large powers of two. In that case adequate array padding is very often able to resolve most conflicts.

The focus of this chapter is on two–dimensional red–black Gauss–Seidel codes on structured grids with constant coefficients. The data locality optimizations described for this restricted case can be applied in the same way to solvers of three–dimensional problems and solvers for variable coefficients. However, some additional issues arise. Three–dimensional problems imply larger data structures and the data structures inherently suffer from cache–interference. Thus, blocking techniques are only effective if sophisticated array padding techniques are applied. Solvers for variable coefficients imply additional data structures which introduce a higher potential of cache–interference among themselves. Thus, sophisticated data layout techniques have to be applied in this case as well to sustain a high performance.

Chapter 6

DiMEPACK: A Cache–Optimized Multigrid Library

In order to demonstrate the applicability of the new data locality optimization techniques described in the previous chapter the optimized red–black Gauss–Seidel smoothers have been integrated in the DiMEPACK multigrid library [KKRW01, KW01].

DiMEPACK is the result of a joint effort of the Lehrstuhl für Informatik 10, University Erlangen–Nuremberg, Germany, and the Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR–TUM), Technische Universität München, Germany. It provides a comfortable C++ user interface which comprises data types for grid functions, operators, multigrid routines, etc. The computational intensive parts such as the smoother and inter–grid transfer operations have been developed as part of this thesis and build the core routines of the library.

The core routines have been implemented in Fortran77 and utilize the data locality optimizations described in the previous chapter, arithmetic optimizations, and new locality improved inter–grid transfer operations which will be described in Section 6.3.2. In order to reduce the amount of conflict misses various array padding heuristics have been integrated within a C++ padding library which is used by DiMEPACK.

DiMEPACK is a C++ library of standard and cache–optimized multigrid implementations for the numerical solution of elliptic partial differential equations (PDEs) in two–dimensions. DiMEPACK[1] can handle constant–coefficient problems on rectangular domains. An overview of the DiMEPACK library is illustrated in Figure 6.1. The parts of the library which are dyed grey, are implemented as part of this thesis and utilize the data locality optimization techniques.

In the following, the functionality of DiMEPACK, arithmetic optimizations, data locality optimizations for inter–grid transfer operations, and the C++ array padding library utilized in DiMEPACK will be described.

[1]DiMEPACK was developed as part of the DiME project (**D**ata local **i**terative **ME**thods). The DiME project was funded in part by the German Science Foundation (Deutsche Forschungsgemeinschaft), research grants Ru 422/7–1,2,3. The library was named after the common practice to add the ending **PACK** to the names of numerical libraries.

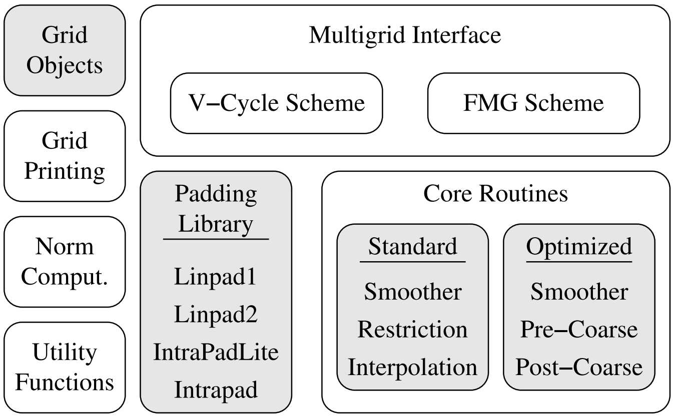

Figure 6.1: DiMEPACK library overview

6.1 Functionality of DiMEPACK

This section only gives a brief introduction to the functionality of DiMEPACK. For a more detailed description see the DiMEPACK User Manual [KKRW01].

DiMEPACK implements both V–cycles and full multigrid (FMG, nested iteration) based on a coarse–grid correction scheme. Full–weighting as well as half–weighting are implemented as restriction operators. The prolongation of the coarse–grid corrections is done using bilinear interpolation. DiMEPACK integrates the optimized Gauss–Seidel resp. SOR smoothers based on a red–black ordering of the unknowns which have been developed in Chapter 5.

DiMEPACK can handle constant–coefficient problems based on discretizations using 5–point or 9–point stencils. DiMEPACK is applicable to problems on rectangular domains where different mesh widths in both space dimensions are permitted. It can handle bothDirichlet and Neumann boundary conditions. Equations for Neumann boundary nodes are obtained by introducing second–order central difference formulae for approximating the external normal derivatives. The arrays containing the boundary data have to be passed as parameters to the library functions.

The user may specify the total number of grid levels to be used. The linear systems on the coarsest grid are solved using LAPACK library routines [ABB$^+$99]. For this purpose the corresponding matrix is split into two triangular factors in the beginning of the computation. If this matrix is symmetric and positive definite, a Cholesky factorization is computed. Otherwise an LU factorization is determined. In the course of the iterative process the coarsest systems are then solved using forward–backward substitution steps.

Various stopping criteria for the multigrid iterations have been implemented which can be specified by the user. In general, the computation stops as soon as either a maximum number of multigrid cycles have been performed or as soon as the discrete L2 norm or the maximum norm of the residual has dropped below a prescribed threshold value.

Before the installation of the DiMEPACK library the user may set an environment variable in order to specify either single precision or double precision as the type of floating–point representation to be used. The choice of single precision arithmetic can significantly speed up code execution. Furthermore, the user may activate or deactivate all data locality optimizations at installation time. Thus, DiMEPACK can be used as standard or locality optimized multigrid library. Once DiMEPACK is installed it can be used like any C++ library. Subroutines which implement appropriate arithmetic resp. data locality optimizations are chosen according to the input parameters at runtime without the knowledge of the user of the library.

6.2 Arithmetic Optimizations

Arithmetic optimizations do not aim at enhancing the cache performance, but at minimizing the number of floating–point operations to be executed, without losing the identity of the numerical results[2]. The DiMEPACK library selects specialized Fortran77 subroutines at runtime according to the specified input parameters. The subroutines are automatically generated at installation time by a macro processor mechanism from a common code set. The following arithmetic optimizations are implemented in DiMEPACK:

- DiMEPACK respects that, if both a 5–point stencil and a Gauss–Seidel smoother are used, the residuals vanish at the black grid nodes. This drastically simplifies the implementation of the restriction operators saving both floating point instructions and load operations.

- If the problem is homogeneous, i.e. if the right–hand side of the linear system corresponding to the finest grid equals 0, DiMEPACK uses dedicated smoothing functions in order to avoid unnecessary memory accesses.

- In order to save multiply operations, both the relaxation parameter of the smoothing routine (if different from 1) and the diagonal entries of the matrices are respected during a preprocessing step for the finest grid and in the course of the residual restrictions for the coarser grids, respectively.

- DiMEPACK saves multiply operations by factoring out identical stencil coefficients whenever possible.

- If the matrix to be factorized on the coarsest grid is symmetric and positive definite, the Cholesky's method is applied instead of computing an LU decomposition. This approximately saves 50 per cent of the corresponding floating–point operations.

[2]Marginal differences in the numerical results may occur due to a reordering of the arithmetic instructions by the compiler. Note that certain arithmetic rules, like for example the law of associativity, do not hold for finite–precision floating–point arithmetic.

6.3 Data Access Transformations

The DiMEPACK library includes a standard multigrid implementation as well as cache–optimized multigrid implementations. Data locality optimizations have been applied to the red–black Gauss–Seidel smoother as well as to the inter–grid transfer operations. The user may specify at installation time whether cache optimized or non–optimized routines should be used.

6.3.1 Smoother Optimizations

The most time consuming part in a multigrid method is the smoother. To guarantee high performance DiMEPACK includes cache optimized red–black Gauss–Seidel smoothers written in Fortran77 which can handle constant–coefficient problems based on discretizations using 5–point or 9–point stencils. The smoothers described in Chapter 5 are extended to handle arbitrary rectangular domains and to allow Dirichlet as well as Neumann boundary conditions. The implemented optimization techniques include the *fusion*, *one–dimensional blocking*, and *two–dimensional blocking* (based on the skewed approach) techniques. DiMEPACK automatically selects an appropriate red–black Gauss–Seidel variant at runtime according to the grid size and basic machine parameters like cache sizes.

6.3.2 Inter–Grid Transfer Optimizations

Within a multigrid algorithm the following operations are performed:

- Pre–smoothing
- Residual calculation and restriction
- Interpolation
- Post–smoothing
- Direct solver

The direct solver is only applied to the problems on the coarsest grid. The coarsest grid in the multigrid context is usually small enough to fit in the L1 or L2 cache. Hence, the direct solver is not discussed in this thesis.

The operations performed in a multigrid context can be divided into *pre–coarse grid* and *post–coarse grid* operations as illustrated in Figure 6.2. Pre–coarse grid operations include all operations which are performed before the algorithm proceeds to the next coarser grid whereas the post–coarse grid operations include the operations which are performed after the correction is determined with the coarser grids.

The pre–smoothing, residual calculation and restriction operation are considered pre–coarse grid operations. After the multigrid method performed a smoothing phase on the

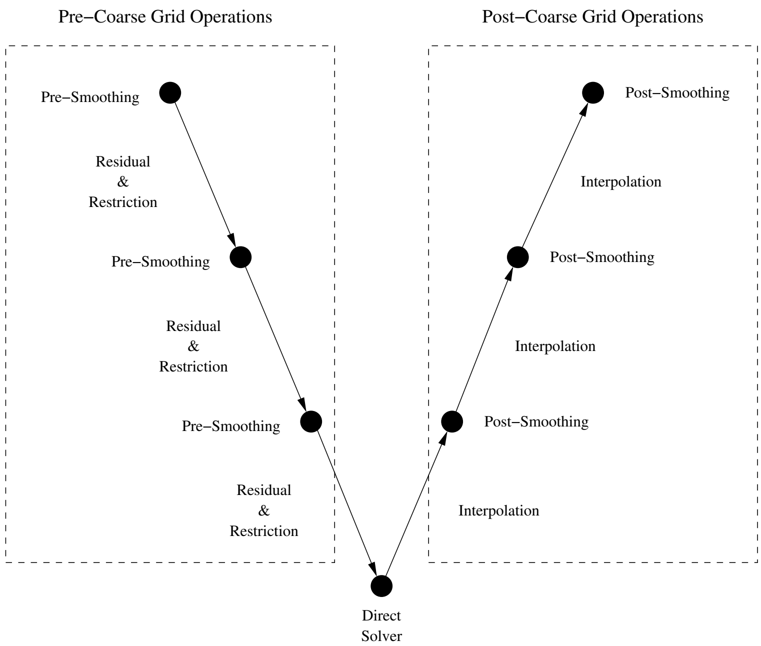

Figure 6.2: Pre– and post–coarse grid operations.

grid and the residuals for all grid points have been calculated, an inter–grid transfer operation is required to transfer the residuals to the next coarser grid. DiMEPACK provides half injection and full–weighting as restriction operations implemented as Fortran77 subroutine. To save an additional global sweep through the finer grid the residual calculation and restriction operation is combined.

Post–coarse grid operations include the inter–grid transfer operation interpolation which propagates the correction of the current approximation to the next finer grid and eventually post-smoothing. DiMEPACK only provides a bilinear interpolation implemented as Fortran77 subroutines. Although post–smoothing is not mandatory usually at least a small number of post–smoothing steps are performed after applying the correction to smooth the new approximation on the finer grid.

The pre- and post-smoothing operations perform independently global sweeps through the grid. Similar, the inter–grid transfer operations involve global sweeps through the fine grid data structure. In contrary to the (non cache–optimized) smoother each inter–grid transfer operation, however, only involves a single sweep through the grid. Thus, data locality optimizations considered for an inter–grid transfer operation individually will only exploit spatial locality.

Nevertheless, there is some potential to improve data locality. All pre–coarse grid resp. all post–coarse grid operations perform individually global sweeps through the same grid. If the grid and the next coarser grid in the case of the pre–coarse grid operations resp. the next finer grid in the case of the post–coarse grid operations do not fit in the cache the data which was loaded into the cache in the process of one operation will not be

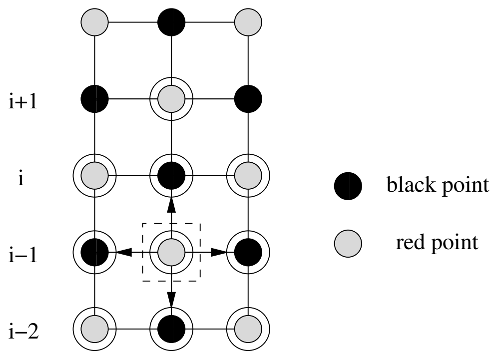

Figure 6.3: Data dependences in pre–coarse grid operations.

reused by the global sweep through the data structure by the following operation. Thus, combining all pre–coarse grid resp. all post–coarse grid operations so that only a single sweep through the grid is performed for each improves data locality. Of course this must be done carefully, so that no data dependences of the underlying algorithm are violated.

DiMEPACK implements melted pre– and post–coarse grid operations for all optimized red–black Gauss–Seidel variants. How the pre– resp. post–coarse grid operations can be combined is explained in the following.

Pre–Coarse Grid Operations

The first pre–coarse grid operation is the smoothing step which includes one or more iterations of the red–black Gauss–Seidel algorithm. After that the residuals are calculated for all nodes. Note, that if a 5–point stencil and the Gauss–Seidel smoother is used the residuals at the black points are guaranteed to be equal to zero. Thus, only the residuals at the red grid points must be calculated.

If a 5–point stencil is assumed the residual calculation of any red node requires that all neighboring black nodes are fully relaxed. Consider the red node in row $i-1$ of the grid part illustrated in Figure 6.3. Assume that each grid point is relaxed exactly once then all neighboring black points will be up to date as soon as the black node in row i directly above the examined red node is up to date. The black node in turn will be fully relaxed as soon as the red node in row $i+1$ directly above it is fully relaxed. Thus, the residual calculation can be handled like an additional red–black Gauss–Seidel iteration.

Once a residual is calculated the contribution of that node can be propagated to the corresponding coarse grid point. The handling of the grid transfer involves a minor coding difficulty since depending on the restriction operation not all fine grid points contribute to the right–hand side of the equation on the next coarser grid. For example, the half–injection operation only uses the residuals of red points on even grid lines (provided the residual of all black nodes is zero).

In the case of a 9–point stencil discretization all neighboring red points must be fully relaxed as well besides the black points. Similarly, this will be true once the red node in

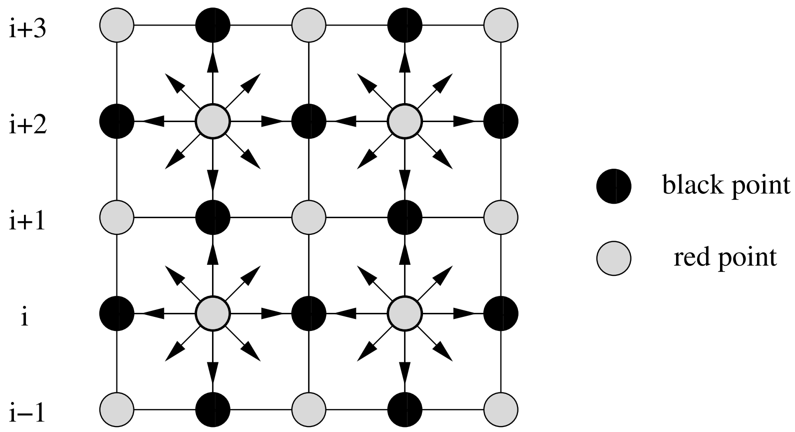

Figure 6.4: Data propagation of the binlinear interpolation.

row $i+1$ directly above the red node in row $i-1$ is fully relaxed. Thus, the 5–point and 9–point stencil case can be handled equally.

Post–Coarse Grid Operations

The first post–coarse grid operation is the interpolation followed by optional post–smoothing. The correction which is stored in a coarse grid point $(i,j)_{coarse}$ located in the interior of the grid is propagated to the fine grid point $(2*i, 2*j)_{fine}$ which is directly above it and to the eight neighboring fine grid nodes as illustrated in Figure 6.4. In the case of Neumann boundaries, for a coarse grid point $(i,j)_{coarse}$ located on a boundary the correction is propagated to the fine grid boundary point $(2*i, 2*j)_{fine}$ directly above it, the neighboring fine grid boundary points, and to the neighboring inner grid points. Hence, fine grid nodes in the interior of the grid like the red node in the middle of row $i+1$ in Figure 6.4 receive data from four coarse grid points represented in the Figure by the thick circle around the fine grid points directly above them.

If a discretization based on a 5–point stencil of the differential operator is used a node can be relaxed as soon as coarse grid data has been applied by the interpolation subroutine to the node itself and its four neighboring nodes. Consider one of the red nodes in row i illustrated in Figure 6.4. The red node can be relaxed as soon as the nodes up to the black node directly above the red node have received the correction from the coarse gird points. In the case of a 9–point stencil, four additional points must receive their coarse grid corrections. Again the red node in row i can be relaxed as soon as all nodes up to the red node directly north east of it received the coarse grid data. Thus, if both cases are combined the relaxation of a red node in row i is must be preceeded by the interpolation of the black node in row $i+1$ directly above it and the interpolation of the red node to the northeast of it. If all red nodes of the grid are processed in that way the black nodes can be relaxed without further interpolation. Similarly to the red–black Gauss–Seidel optimizations, however, the boundaries of the grid must be treated separately.

6.4 Data Layout Transformations

As demonstrated in Chapter 5 the blocked red–black Gauss–Seidel implementations suffer from cache–interference. *Array padding* proved to be able to reduce the number of *conflict misses* dramatically on cache–based architectures. The reduction of conflict misses, however, depends on a proper selection of the padding size. An exhaustive search for a decent padding might be applicable if a program is fine tuned for a certain architecture, however, for a library like DiMEPACK or a compiler this approach is not applicable.

Tseng et al. [RT98a, RT98b] introduced several padding heuristics which can be used in a compiler to identify bad array layouts. The heuristics use information such as array descriptions, cache size, cache line size, and in some cases information about array references. Once a bad array layout is identified by the heuristics the compiler applies array padding, i.e. the size if the array is changed, and heuristics are used again to check the new array layout. This process is repeated until a good array padding size is found. The following intra and inter array padding heuristics have been implemented in a C++ class library from the descriptions published in [RT98a, RT98b] as part of this thesis:

- Linpad1
- Linpad2
- IntraPadLite
- IntraPad
- InterPadLite
- InterPad

LinPad1, *LinPad 2*, *IntraPadLite*, and *IntraPad* are used for intra array padding. Contrary, *InterPadLite* and *InterPad* are used for inter array padding. For multigrid methods intra array padding is sufficient if the sizes of the paddings for the different arrays are ensured to be different. Therefore, the explanation of the heuristics is restricted to the intra array padding heuristics used within DiMEPACK.

LinPad1 and LinPad2 are very simple heuristics which only require information about the array, cache size (C_s), and cache line size (L_s) to determine whether an array size is likely to produce self interference misses. In principle, they avoid column sizes (Col_s) which have a large power of two as a factor. LinPad1 assumes that array padding is necessary if $(j * Col_s \bmod C_s) < 0$ for a small j because only the first C_s/j multiplies of Col_s will be mapped to distinct locations in the cache. The LinPad2 heuristics assumes that array padding is necessary if $(j * Col_s \bmod C_s) < L_s$ for a small j. Both heuristics can be applied even if no information about the program behavior is available. However, they are typically only able to identify pathological cases.

Like the LinPad heuristics IntraPadLite only requires information about the size of the array and the cache size to determine whether an array size causes conflict misses. The accuracy should therefore be similar to LinPad1 and LinPad2. IntraPadLite rejects the size of a two–dimensional array if $(Col_s \bmod C_s) < M$ or $(2 * Col_s \bmod C_s) < M$. M is the minimum distance of separation between nearby columns which should be ensured by the heuristics. Tseng et al. stated that $M = 4$ should be sufficient in most cases.

A more sophisticated padding heuristics is IntraPad. In addition to the array and cache information it also utilizes information about the array references within a loop nest. Besides severe interference which is detected by the heuristics described so far, IntraPad is able to detect semi–severe interference, i.e. interference which does not occur in every loop iteration but in a high percentage of all iterations. IntraPad tests any combination of two array references within a loop nest. If they access two array elements which are mapped to the same cache line the current array size is rejected and (different) padding is applied. To simplify the address calculation for the array references the heuristic requires uniformly generated array references [GJG88]. Thus, once the two array references are linearized the array index variable terms cancel leaving only the constants of the array references.

All heuristics are aimed at one level of the memory hierarchy. DiMEPACK allows the user to specify the cache level for the array padding heuristics with the help of environment variables. The information is taken from the environment variables at runtime to optimize the layout of the dynamically allocated grid structures.

6.5 DiMEPACK Performance Evaluation

6.5.1 Smoother Performance

The DiMEPACK multigrid code uses the red–black Gauss–Seidel algorithm as smoother component. In order to demonstrate the applicability of the optimization techniques described in this thesis, the DiMEPACK smoothers have been optimized by the fusion, one–dimensional blocking, and skewed two–dimensional blocking technique.

The red–black Gauss–Seidel algorithms used in DiMEPACK, however, are more general than the algorithms used for the demonstration of the concept in Chapter 5. They allow the handling of both Dirichlet and Neumann boundary conditions as well as rectangular grids. The algorithms used in Chapter 5 are restricted to Dirichlet boundary conditions and square grids. Furthermore, DiMEPACK includes red–black Gauss–Seidel variants which utilize arithmetic optimizations to reduce the number of floating point and load operations to be executed. Thus, DiMEPACK includes several specialized red–black Gauss–Seidel codes which are invoked at runtime according to the input parameters of the library.

Figure 6.5 illustrates the performance of the standard[3] and cache optimized DiMEPACK red–black Gauss–Seidel smoothers without applying any arithmetic optimizations

[3]The standard red–black Gauss–Seidel smoother in DiMEPACK is comparable to the red–black Gauss–Seidel smoother described in Chapter 5 after applying the array transpose technique. Thus, the pathological case of a very bad data layout is avoided in DiMEPACK.

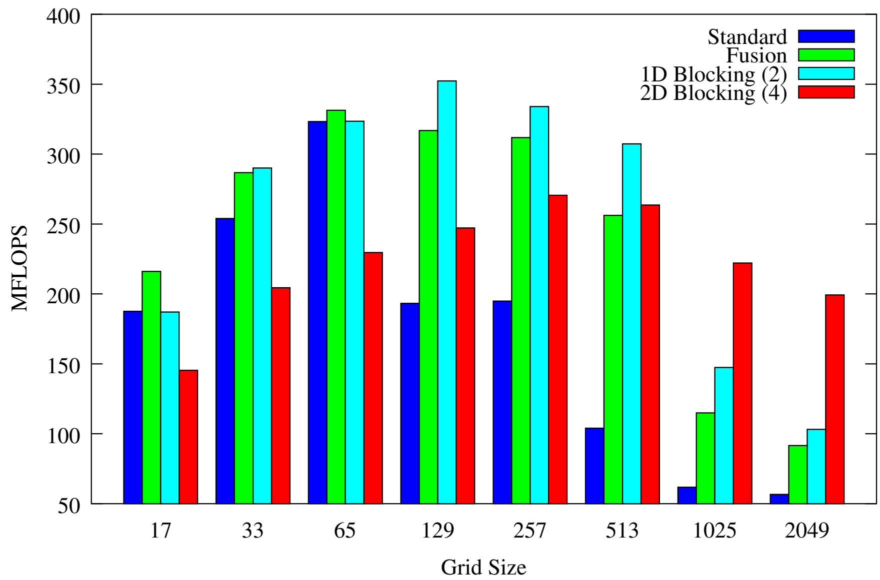

Figure 6.5: DiMEPACK red–black Gauss–Seidel smoother performance.

on a Compaq PWS 500au[4]. The red–black Gauss–Seidel smoothers use a discretization based on a 5–point stencil and double precision floating point arithmetic. The MFLOPS rates for the small grid sizes are slightly lower than the MFLOPS rates of the smoothers described in Chapter 5. The reason for this is that the handling of the grid boundaries is more complicated. Thus, branch miss predictions and register dependences limit the performance for the small grid sizes. For the larger grid sizes, however, similar MFLOPS rates are achieved (compare Figure 5.17 and Figure 6.5). Thus, the execution of the smoother component is accelerated by almost a factor of four for the large grid sizes on the Compaq PWS 500au.

The data locality optimization techniques described in this thesis do not change the amount of floating point operations to be executed by the smoother algorithm. Thus, an improvement in the MFLOPS rate is equivalent to a reduction in program runtime. However, this is true for the arithmetic optimizations used within DiMEPACK. The arithmetic optimizations reduce the number of floating point and load operations to be executed by exploiting special cases of the input parameters. DiMEPACK includes three arithmetic optimizations which affect the code of the red–black Gauss–Seidel algorithm. First, if the problem is homogeneous, i.e. if the right–hand side of the linear system corresponding to the finest grid equals 0, one load operation and one floating point operation is saved per relaxation. Thus, the right–hand side of the equation is not required effectively halving the working set of the smoother algorithm. Second, if the relaxation parameter of the smoothing routine is equal to one, i.e. a standard Gauss–Seidel and not a successive over relaxation (SOR) method is used, two floating point operations and eventually one load

[4]Compaq PWS 500au with 500 MHz Alpha 21164 and 4 Mbyte L3 cache. For compilation the compiler options “-O5 -fast -tune host -arch host” of the Compaq Fortran77 compiler (version 5.10) were used.

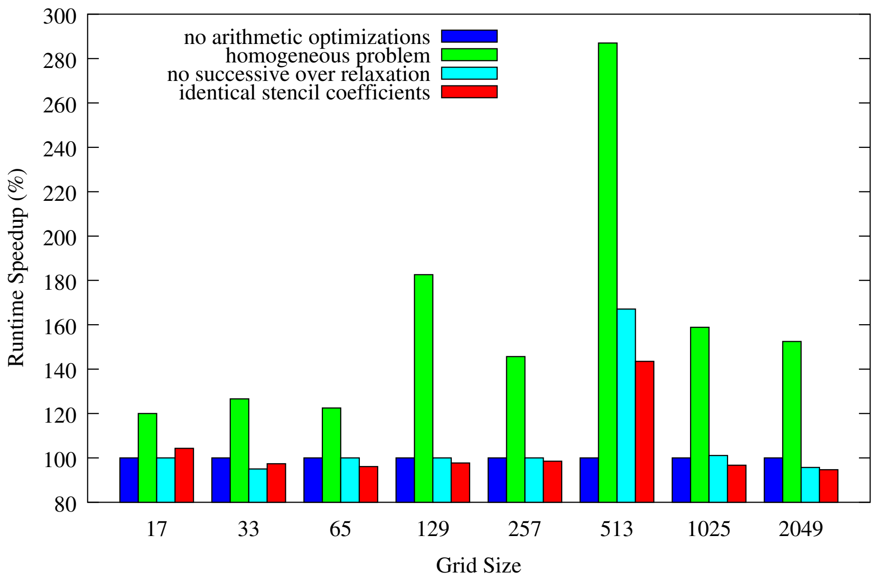

Figure 6.6: Runtime speedup of standard red–black Gauss–Seidel smoother obtained with arithmetic optimizations.

operation can be saved per relaxation. Note, however, that the compiler might keep the value of the relaxed node in a register as described in Section 5.1.2 so that the load operation is not executed even without applying the arithmetic optimization. Third, DiMEPACK saves multiply operations by factoring out identical stencil coefficients. Thus, in the 5–point stencil case three and in the 9–point stencil case six floating point multiplies are saved per relaxation. Finally, combinations of the three optimizations are possible.

Figure 6.6 shows the runtime speedup of the standard red–black Gauss–Seidel algorithm in per cent obtained by applying the three arithmetic optimizations described above one by one. Only the first optimization achieves a significant runtime speedup for all grid sizes. The runtime of the red–black Gauss–Seidel codes for most grid sizes after applying the other arithmetic optimizations is only slightly lower or in some cases even slightly higher. The first arithmetic optimization is able to improve the performance of the red–black Gauss–Seidel smoother since it eliminated the accesses to the right–hand side of the equation. Thus, it reduces the amount of data to be kept in cache simultaneously. The other two arithmetic optimizations mainly reduce the number of floating point operations to be executed per relaxation of one nodes. As explained in this thesis, however, the number of floating point operations to be executed is not the limiting factor but the number of load and store instructions to be executed. Consequently, no significant performance improvement is noticeable. Furthermore, the mathematical term to be calculated for the relaxation of one node contains several multiplies which can be performed concurrently. Thus, if the microprocessor implements several floating point execution units the calculation of them will be relatively cheap. After factoring out the identical stencil coefficients, however, the new mathematical term only contains one (resp. two multiplies in the 9–

Grid Size	no arithmetic opt. Std.	Max.	Speedup	homogeneous ($f = 0$) Std.	Max.	Speedup
17	187.6	216.1	1.15	205.0	240.3	1.17
33	253.9	290.0	1.14	286.9	341.1	1.19
65	323.3	331.3	1.02	361.1	423.1	1.17
129	193.3	352.3	1.82	319.8	490.0	1.53
257	194.9	354.3	1.81	254.4	464.1	1.82
513	104.0	325.2	3.13	258.6	484.0	1.87
1025	61.8	222.1	3.59	88.9	308.8	3.47
2049	56.6	199.3	3.52	78.0	267.0	3.42
Grid Size	**$\omega = 1.0$ (no SOR) Std.**	**Max.**	**Speedup**	**identical coefficients Std.**	**Max.**	**Speedup**
17	149.3	175.0	1.17	143.4	143.7	1.00
33	194.1	242.0	1.25	178.0	215.9	1.21
65	260.0	311.8	1.20	222.8	242.4	1.09
129	153.5	315.0	2.05	137.0	244.1	1.78
257	157.0	317.7	2.02	137.0	253.7	1.85
513	132.8	290.8	2.19	87.4	240.4	2.75
1025	50.0	182.7	3.65	43.1	174.0	4.04
2049	45.9	158.8	3.46	39.7	157.3	3.96

Table 6.1: Standard DiMEPACK smoother performance (column “Std.”) compared to the best smoother performance achieved with data locality optimizations (column “Max.”) for different red–black Gauss–Seidel smoother variants.

point stencil case) which can be performed after all values of the neighboring nodes have been added up. Thus, in the latter case the potential ILP is lower.

Table 6.1 summarizes the performance of the smoother variants with and without data locality optimizations. The MFLOPS rates of the red–black Gauss–Seidel variants vary significantly. However, for all variants the data locality optimizations techniques are able to achieve comparable performance improvements.

6.5.2 Multigrid Performance

The red–black Gauss–Seidel smoother is the most time consuming component of the DiMEPACK multigrid code. Typically, the smoother consumes 70 to 90 per cent of the runtime of the whole multigrid. The percentage depends on the number of pre– and post–smoothing steps which are applied in the multigrid code. Improving the performance of the red–black Gauss–Seidel smoother will result in a speedup of the whole multigrid program which can be calculated with Amdahl’s law.

Figure 6.7 summarizes the MFLOPS rates for a a multigrid algorithm performing V–cycles with two pre– and two post–smoothing steps (V2,2) on a Compaq PWS 500au.

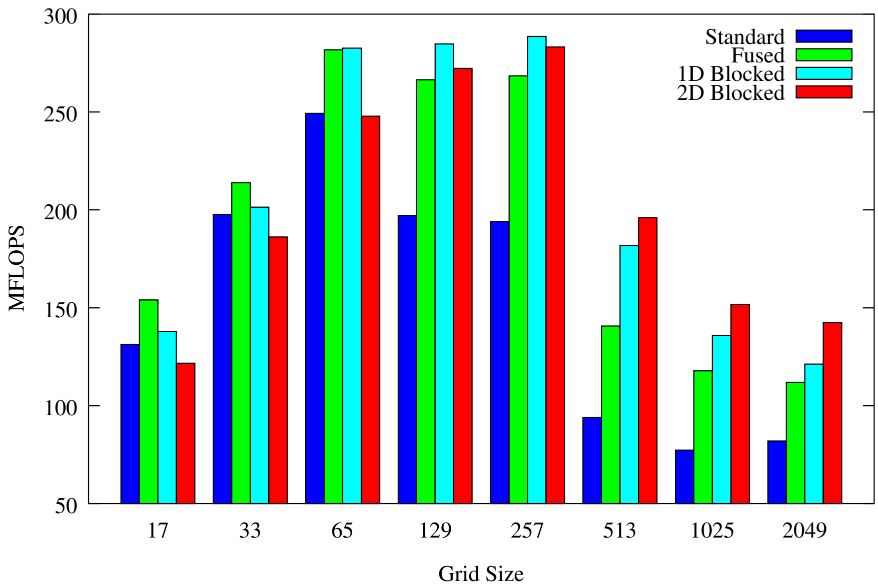

Figure 6.7: DiMEPACK multigrid performance with different smoother optimizations.

The multigrid code was configured to use a discretization based on a 5–point stencil, the full–weighting restriction operation, no arithmetic optimizations, and double precision floating point arithmetic. The MFLOPS rates of the multigrid code using a standard red–black Gauss–Seidel smoother is shown in comparison to the multigrid code with data locality improved red–black Gauss–Seidel smoothers.

To measure the data shown in Figure 6.7 the DiMEPACK multigrid code was forced to use one data locality optimization technique for all grid sizes. Normally, DiMEPACK automatically selects the red–black Gauss–Seidel variant obtained by the data locality optimization which is best for each grid size. Thus, red–black Gauss–Seidel codes which result from different data locality optimizations will be used for different grid sizes.

Table 6.2 shows the MFLOPS rates for a multigrid code executing two resp. four pre– and post–smoothing steps with the same configuration (5–point stencil, full–weighting restriction operation, no arithmetic optimizations, and double precision floating point arithmetic) automatic data locality optimization selection on a Compaq PWS 500au. The MFLOPS rates for the same multigrid code which, however, uses smoothers that utilize a combination of the second and third arithmetic optimizations are summarized in Table 6.2. Both tables show similar speedups for the multigrid code obtained by the data locality optimizations. Especially for the large grid sizes a remarkable speedup of approximately a factor of two for the whole multigrid algorithm is achieved. The speedups for V(4,4) multigrid, thereby, is higher since the fraction of the runtime spent in the smoother component is larger.

To conclude the DiMEPACK performance evaluation the runtime of the DiMEPACK multigrid code, the execution times (in seconds) of a multigrid code (5–point stencil, full–weighting, arithmetic optimizations enabled, and double precision arithmetic) performing

Grid Size	V(2,2) multigrid			V(4,4) multigrid		
	Std.	Opt.	Speedup	Std.	Opt.	Speedup
17	131.2	150.4	1.15	140.4	160.1	1.14
33	197.7	211.7	1.07	204.1	217.4	1.07
65	249.3	275.2	1.10	261.2	300.5	1.15
129	197.2	282.5	1.43	202.5	307.9	1.52
257	194.1	287.2	1.48	195.0	320.7	1.65
513	93.9	179.9	1.92	84.2	228.9	2.72
1025	77.3	151.1	1.95	75.3	183.6	2.44
2049	82.0	142.4	1.74	74.8	168.8	2.26

Table 6.2: MFLOPS rates of a multigrid code using two or four pre– and post–smoothing steps, respectively. No arithmetic optimizations were applied to the smoother. Columns "Std." show the performance with a standard red–black Gauss–Seidel smoother whereas columns "Opt." show the performance of the multigrid code with data locality optimized red–black Gauss–Seidel smoothers.

Grid Size	V(2,2) multigrid			V(4,4) multigrid		
	Std.	Opt.	Speedup	Std.	Opt.	Speedup
17	85.4	92.5	1.08	83.8	95.7	1.14
33	122.1	130.9	1.07	118.8	128.5	1.08
65	147.9	173.5	1.17	149.7	177.0	1.18
129	114.9	170.2	1.48	110.9	175.3	1.58
257	110.7	169.4	1.53	105.9	181.1	1.71
513	56.6	111.3	2.00	48.7	131.9	2.71
1025	51.0	100.5	1.97	45.0	118.3	2.63
2049	61.5	106.6	1.73	50.2	119.6	2.28

Table 6.3: DiMEPACK performance overview with typical arithmetic optimizations.

Machine	Grid Size							
	17	33	65	129	257	513	1025	2049
Compaq PWS 500au	0.017	0.018	0.021	0.031	0.074	0.278	1.177	4.792
Compaq XP1000	0.010	0.011	0.012	0.019	0.042	0.141	0.605	2.392
Pentium 4 PC	0.010	0.010	0.010	0.010	0.040	0.120	0.500	2.220

Table 6.4: DiMEPACK multigrid runtime in seconds for one V(2,2) V–cycle.

Grid Size	V(2,2)			V(4,4)		
	Std.	Opt. rbGS	Opt. Inter	Std.	Opt. rbGS	Opt. Inter
17	131.2	150.4	112.6	140.4	160.1	126.4
33	197.7	211.7	127.1	204.1	217.4	154.1
65	249.3	275.2	175.3	261.2	300.5	237.5
129	197.2	282.5	191.0	202.5	307.9	231.5
257	194.1	287.2	203.9	195.0	320.7	243.4
513	93.9	179.9	157.6	84.2	228.9	191.8
1025	77.3	151.1	123.0	75.3	183.6	130.5
2049	82.0	142.4	113.8	74.8	168.8	117.8

Table 6.5: MFLOPS rates for DiMEPACK V–cycles (two resp. four pre– and post–smoothing steps) with optimized inter–grid transfer operations compared to a standard multigrid and a multigrid with optimized red–black Gauss–Seidel smoothers. Arithmetic optimizations were disabled.

a single V–cycle with two pre– and two post–smoothing step for various grid sizes on a Compaq PWS 500au (500 MHz), a Compaq XP1000 (500 MHz), and a Pentium 4 PC (1.5 GHz) are shown in Table 6.4.

6.5.3 Performance Impact of the Inter–Grid Transfer Optimization

The DiMEPACK includes data locality improved inter–grid transfer operations. The optimization techniques combine all pre– resp. post–coarse grid operations to a single operation. Thus, instead of one sweep through the data structure for the (optimized) smoother and an additional for the inter–grid transfer operation only a single sweep is performed which combines both operations.

The impact of these optimizations on the MFLOPS rates of a multigrid code is illustrated in Table 6.5. For small grid sizes the performance of the whole multigrid method is actually deteriorated. For the grid sizes larger than 129×129 the performance for the multigrid code is higher than the performance of a standard multigrid code. However, for all grid sizes the performance of the multigrid code with combined operations is lower than for the multigrid code with optimized red–black Gauss–Seidel smoothers alone.

The reason for this is that the more complex code of the combined operations slows down the smoother part due to higher register dependences and branch mispredictions. Since the smoother part is the most time consuming part of the multigrid the loss in performance in the smoother component outmatches the slightly better data locality for the inter–grid transfer operations.

6.5.4 A Case Study: Chip Placement

In order to test DiMEPACK with a more complex environment, DiMEPACK was integrated into a global cell placement method for VLSI circuits developed at the Institute of Electronic Design Automation (LEA), Technische Universität München [EJ98].

The different objectives involved in the cell placement can be mapped to a wire length minimization problem under the constraints that cells are not allowed to overlap. Shorter wires implicate a better area usage, faster circuit switching, and reduced power consumption. The method developed at LEA is a force–directed iterative algorithm which models the objectives of the placements with forces and force fields. In each step of the placement algorithm a force field for the current placement is calculated which is described with a Poisson equation. The forces indicate in which way the current placement must be changed to obtain a better placement.

The Poisson equation usually requires the solution of a system of linear equations with several millions of unknowns on rectangular domains. The original solver used by the chip placement code was based on a fast Fourier transformation. The fraction of time spent in the FFT solver is approximately 50 per cent of the total runtime. Thus, according to Amdahl's law a speedup of at most two can be expected when the FFT solver is replaced by the DiMEPACK multigrid solvers. In general, multigrid method are an efficient approach for the solution of such problems. The high memory requirement of the problem, however, makes the usage of cache–optimized solvers indispensable.

The DiMEPACK multigrid solvers were able to accelerate the solution of the Poisson's equation for the three biggest problems of the LayoutSynth92 benchmark set[5] by a factor of 1.19 on average. The chip placement code is further accelerated since less iteration of the algorithm are required. The reason for this is the larger functionality of the DiMEPACK multigrid code compared to the FFT code. An integration of the multigrid code which exploits the full functionality of DiMEPACK as well as further benchmarks is subject to further research.

6.6 Related Work

With the still growing gap between microprocessor and memory speed, locality optimizations in general and especially data locality optimizations are considered crucial for good application performance on modern computer systems. The definition of data locality stems from an early work by Wolf and Lam [WL91]. Several data access transformations and data layout transformations have been proposed to optimize simple loop nest constructs. More information about research done in that area can be found in Chapter 4.

First considerations of data locality optimizations for iterative methods have been published by Douglas [Dou96]. They include simple data locality optimizations comparable to the *fusion* technique described in this thesis. The discussion of optimizations for an entire multigrid algorithm, however, is limited to a concept draft. More recent work done

[5]The LayoutSynth92 benchmark suit is available at http://www.cbl.ncsu.edu/CBL_Docs/lys92.html.

by Douglas et al. and Hu [DHK^{+}00b, Hu00] focuses on optimizations for multigrid methods on unstructured grids. The focus of data locality optimizations for unstructured grids is slightly different to optimization on structured grids, since the data structures involved with unstructured grids are not array based but pointer data structures. Thus, minimizing the latency involved with accessing data through a chain of pointers becomes dominant. Hu [Hu00] proposes a relatively inexpensive grid reordering and blocking strategies based on the new ordering to improve locality.

Research in a similar area has been reported by Keyes et al. [GKKS01]. They used manually applied data locality optimizations for complex iterative methods on unstructured grids and achieved a significant performance improvement with array merging, blocking, and node reordering techniques. They proved the efficiency of their techniques by presenting detailed miss rates for caches and TLB.

Tseng et al. [RT00] demonstrated how tile size and padding size selection can be automated for three–dimensional multigrid codes. The work is based on the research presented in this thesis.

Kowarschik [Kow01] and Pfänder [Pfä00] applied data layout and data locality optimizations for variable coefficient multigrid. The data access transformations are based on the work in this thesis. However, they demonstrated that an appropriate data layout has to be ensured by array merging and array padding to ensure high performance.

Genius et al. [GL01] recently proposed an automatable method to guide array merging for stencil based codes based on a meeting graph method. However, the technique is only applicable to the innermost loop in a loop nest.

Quinlan et al. [BDQ98] introduced the term *temporal blocking* for stencil based operations. Temporal blocking is a loop transformation comparable to the loop transformation *tiling*. They applied temporal blocking to the Jacobi method but did not show how temporal blocking can be applied to red–black Gauss–Seidel or multigrid methods.

Finally, recent work by Sellappa et al. [SC01] is based on the temporal blocking idea. They present a two–dimensional blocking strategy for the Gauss–Seidel method based on a standard and red–black ordering of the unknowns. The approach is comparable to the square two–dimensional blocking technique described in this thesis. They do neither consider data locality optimizations for inter–grid operations nor data layout optimizations.

6.7 Summary

The new data locality optimization techniques for the red–black Gauss–Seidel method build the core routines for the DiMEPACK multigrid library. They provide a cache efficient implementation of the smoother component of a multigrid method. Furthermore, new cache–aware inter–grid transfer operations have been introduced and implemented within DiMEPACK. In order to avoid conflict misses introduced by interference a good data layout is ensured with padding heuristics. Finally, the robustness and efficiency of the DiMEPACK multigrid solvers has been demonstrated by means of a chip placement code.

The DiMEPACK library provides an easy to use interface to the sophisticated data locality optimizations developed in this thesis. Thus, existing solver packages for PDEs can be replaced easily by the more efficient and cache–aware DiMEPACK multigrid codes. Furthermore, the DiMEPACK routines can be used as efficient core elements to build more sophisticated PDE solver packages if the functionality of DiMEPACK is not sufficient.

Chapter 7

Tool Support for Data Locality Optimizations

In this chapter, the tool support for data locality optimizations in the software development process is discussed based on the experience gathered during the performance analysis of multigrid methods. Existing performance analysis tools such as program profiling, hardware counter based tools, simulation tools to detect performance bottlenecks are described and discussed in respect to data locality issues.

These tools usually aggregate performance data over time into summary information like miss rates. This type of information, however, is not able to capture the dynamic nature of caches. Thus, existing cache visualization tools are introduced and new memory hierarchy visualization techniques are proposed.

Finally, program transformation tools are discussed which assist the programmers with means to apply cache optimizations to their programs.

7.1 The Software Development Process

There is a large number of existing software development approaches and many of the are actually in use. Introducing even a part of them is not scope of this thesis. Most of the software development approaches, however, include an analysis which leads to a more or less formal specification of the problem. From that specification a design is evolved which is then implemented in a high level programming language. The program represented in a high level programming language is then translated to a machine language program which can be executed on a computer. In most cases the programmers, nowaday, expects that their programs are automatically translated to efficient machine language programs by a compiler system. The task of a compiler system, therefore, does not only include a direct translation of high level constructs to machine instructions but also a transformation of the code which ensures a fast execution on the target machine.

The transformations traditionally include low level optimizations like reordering and grouping of machine instructions. However, high level transformations which rearrange the original high level program become more and more important. The targets of the transformation are manifold and include the reduction of code size, execution time, memory requirement, and the number of executed instructions. Among the targets data locality optimization techniques have become of particular importance for the execution time of programs on RISC based architectures with deep memory hierarchies. In general, the programmer expects that such optimizations are applied fully automatically without any interaction. Unfortunately, the current state of the art in compiler technology does not allow fully automatic optimization for complex programs, i.e. manual interaction of the programmer will be necessary to achieve a good data locality. Thereby, the programmer faces a series of difficult problems.

First, the fact that the program only exploits a fraction of the available peak performance of a machine is not directly apparent. Not many programmers actually know how many operations are executed by their programs nor do they know how many operations can be executed by the machine in a certain time. Typically, a program will only be perceived as too slow if certain application dependent real time constraints cannot be met.

Second, once it became apparent that the program is slow the performance bottleneck must be identified. In many case nowadays the performance is limited by the latency and bandwidth of the main memory but other issues like register dependences or delays due to branch mispredictions might be of importance as well. Depending on the performance bottleneck completely different optimization strategies will be necessary.

As soon as data accesses have been identified as the main performance bottleneck, the basic cause for the poor memory performance must be determined so that code or data layout transformations can be chosen which results in a performance improvement. Applying a loop blocking strategy, for example, will not necessarily lead to a performance improvement when the data within the block causes conflict misses among themselves. Similarly, data layout transformations like array padding will not lead to a significant performance improvement if the majority of the cache misses are capacity or cold misses.

Finally, the code or data layout transformation must be applied to the high level program. Even if the code transformations themselves might be easy additional constraints can make the task for the programmer hard. For example, the transformation should typically keep the semantics of the program unchanged. The more complex the program, the harder it will be in general to restructure the program without violating data dependences. Thus, additional testing or verifying is required after the code has been changed to ensure that no new bugs were introduced by the transformations.

All of the problems mentioned above make the task for the programmer harder and consequently more expensive. Thus, the typical solution taken if a program is too slow is to buy a faster machine instead of spending money for program optimization. If this is not feasible, typically, a parallel version of the program or a hand tuned machine program is implemented. Data locality optimizations are rarely applied although they potentially can speed up the execution of programs by a multiple. The reason for this may be the lack of appropriate data locality performance analysis and code transformation tools.

7.2 Performance Analysis Tools

The principle concept of the memory hierarchy hides the existence of the caches within the hierarchy. Data accesses satisfied by the cache are observed by the programmer, the user of the program, or by a performance analysis tool as absolutely similar to main memory accesses except that the time required to deliver the data differs. This makes data locality optimizations in general difficult and limits the scope of performance analysis tools.

To allow performance profiling regardless of this fact, many microprocessor manufacturers add hardware counters to the CPUs for events like cache accesses, cache hits, or cache misses. Another approach is based on program instrumentation. The information gathered by program information, however, is limited to information like address traces or timing information. Both approaches have the drawback that tools are limited to a certain platform.

A solution for that problem is offered by cache or machine simulation. Based on address traces a virtual machine or a virtual memory hierarchy is simulated. Thus, events like cache hits, misses, etc. can be gathered. Address traces tend to be very large and simulation is typically very time consuming compared to a normal program execution. Thus, the cache resp. machine models are simplified to reduce simulation time. Consequently, the results are often not precise enough to be useful.

All tools have in common that they provide summarized information about the program behavior like hit rates or stall times. Sometimes this information can be attributed to source code lines. A dynamic presentation of the information with the possibility of a navigation in the time frame of the program execution, however, is not intended.

Visualization and animation of the program flow offers the possibility to display dynamic information. Visualization can also give the programmer more insight into the cache behavior of his program. Unfortunately, only some experimental tools with limited functionality are available. In the following, existing performance analysis tools will be introduced.

7.2.1 Performance Profiling

Program Instrumentation

The traditional performance profiling is based on program instrumentation. Calls to a monitoring library are inserted into the program to gather information for small regions of the program. The library routines may include complex programs themselves like simulators or only modify counters. Instrumentation is used, for example, to determine the fraction of the CPU time spent in certain subroutines of the program. Since the cache is not visible for the instrumentation code the information about the memory behavior which can be gathered by such codes is limited to address traces.

gprof [FS98] and *hiprof* [Com01] tools offer *flat profiles* and *call graph profiles*. The flat profile shows the total amount of time a program spent executing each function. The call graph profile shows how much time was spent in each function and its children. From this information functions can be identified which themselves do not use much execution time but called other functions that do use unusual amounts of time. The information can be used to identify functions which are worth to optimize but do not reveal specific memory bottlenecks.

A different profiling approach uses the path profiling tool *PP* [BL96]. A path profile records the execution frequencies of acyclic paths in a routine instead of measuring the frequencies of events in basic blocks. Path profiling is able to precisely identify heavily executed paths in a program. The information can be used to guide compiler optimizations.

Besides the profiling tools instrumentation tools resp. libraries like *Atom* [ES95] or *EEL* [LS95] offer a relatively simply way to insert additional code into an executable. The additional code is executed when an event of a certain type happens while executing the original code like function calls or execution of certain instructions. The inserted code can for example be used to generate address traces for an external cache simulator or directly implement a small simulator.

Hardware Counters

Performance counters are part of many modern microprocessors. They gather relevant events of the microprocessor for performance analysis without affecting the performance of a program. The information which can be gathered by the performance counters varies from platform to platform. Typical events are cache misses, cache hits, cache accesses, cycles, instruction issues to mention some. Unfortunately, many vendors do not provide tools to access the hardware counters of their microprocessors.

One exception is the Compaq (formerly Digital) Continuous Profiling Infrastructure (*DCPI* [ABD$^+$97]) for the Alpha workstations. *DCPI* is based on periodic sampling of the whole system using the Alpha performance counter hardware. Tools are provided to analyze profiles and produce a breakdown of the CPU time spent in each executable image, and in each procedure within an executable. In addition, detailed information is produced showing the total time spent executing each individual instruction in a procedure. Additional analysis tools also determine the average number of cycles taken by each instruction, the approximate number of times the instruction was executed, and the possible reasons for any cycles spent not executing instructions like for example waiting for data to be fetched from memory. With the ability to map hardware counter events to individual statements in a source code *DCPI* provides a very useful interface to performance counters.

A less platform dependent approach is taken by the Performance Counter Library (*PCL* [BM00]). *PCL* allows the access to hardware performance counters on many microprocessors through a uniform interface with low overhead. The application interface supports basic start and stop functionality of performance counting and query functionality for hardware counters. The functionality is available through command line tools and procedure calls in C, C++, Fortran, and Java. The command line tool only provides summarized information for a complete program execution. No automated tools are provided to account events to source code lines. In principle, however, the programmer can use the procedure call interface to realize fine grained profiling. *PCL* is available for systems with Compaq, Sun, SGI, IBM, and Intel microprocessors.

A similar approach is pursued by the *PAPI* project [BDG$^+$00]. The *PAPI* project proposed a standard cross–platform library (API) for accessing hardware performance counters and a standard set of hardware events. The standard API has been implemented in a library for most of the currently available systems. The library provides two interfaces to the underlying counter hardware: a high level interface which provides the functionality to start, stop, and query hardware counters similar to the *PCL* interface and a low level interface which provides the basis for source level performance analysis software. Furthermore, utility routines layered on top of the *PAPI* library are provided to allow a dynamic runtime controlling of the library via socket connections.

7.2.2 Simulation

Cache Simulation

Tycho [Hil87, HS89] is a trace-driven cache simulator that can simulate many alternative configurations of one cache level like direct–mapped, set–associative, and fully–associative caches with one pass through an address trace to produce a table of miss ratios for all caches. The cache parameters which can be varied, however, are severely restricted. For example all caches must have the same cache line size, do no prefetching, and use LRU replacement. *Tycho* and the speed improved version *TychoII* are part of the *WARTS* tool set[1].

The *DineroIII* cache simulator is also part of the *WARTS* tool set. Furthermore, it is part of [HP96]. *DineroIII* is a trace-driven cache simulator which evaluates only one cache at a time. In exchange it produces more performance metrics like for example traffic to and from memory and allows more cache design options like write policy, sub block placement, replacement strategies, or prefetching schemes to be varied.

Cheetah [SA93] is a single–pass cache simulation package which can simulate a range of cache configurations including direct–mapped, set–associative, and fully–associative caches. The input to cheetah is a memory address trace. In each simulation run *Cheetah* outputs the miss ratios of several cache configurations. Besides different degrees of associativity *Cheetah* can simulate caches under LRU and OPT replacement strategies. OPT (also known as MIN [Bel66]) uses future knowledge to select a replacement, i.e. the memory block which will be referenced the furthest in the future is selected for replacement. Thus, the replacement policy is optimal for any set of references. The capabilities of cheetah are a superset of those of Tycho.

All cache simulators described above assume a relatively conservative cache model with one unified or split cache per level of the memory hierarchy. A less conservative cache model is used by the *mlCache* [TRTD98] simulation tool. *mlCache* is an

[1]The WARTS tool set is currently available at http://www.cs.wisc.edu/ larus/warts.html.

event–driven, timing–sensitive simulator of multi–lateral cache designs [RDC$^+$94, Jou90, RD96]. The simulator, furthermore, uses the *Latency Effects* [TD96] cache timing model to take memory latencies into account when calculating miss rates. Thus, besides regular hits and misses also delayed hits are simulated. Delayed hits are accesses to data currently being delivered to the cache because of an earlier miss to the same cache block. The cache models used by *Tycho*, *DineroIII*, and *Cheetah* assume that data arrives instantly. Consequently, delayed hits are not present in these models. The events for the simulator can be provided by traces or by a machine simulator like SimOS [RHWG95] or SimpleScalar [BA97].

The simulation systems described so far produce summarized cache information of the whole execution of a program (trace). The output typically consists of cache miss and hit rates statistics. The *CPROF* [LW94] system couples a trace–based uniprocessor cache simulator with source code annotation. Thus, *CPROF* identifies the source lines and data structures that cause frequent cache misses. Therefore, the source lines and data structures are annotated with the appropriate cache miss statistics which are displayed with a graphical user interface. Beside quantitative data the annotations also include a classification of the cache misses as compulsory, capacity, or conflict.

Machine Simulation

SimOS [RHWG95] is a simulation environment designed for the efficient and accurate study of both uniprocessor and multiprocessor computer systems. *SimOS* currently provides models of the MIPS R4000 and R10000 and Digital Alpha processor families. In addition to the CPU, caches, multiprocessor memory busses, disk drives, Ethernet, consoles, and other devices commonly found in a computer system are simulated. *SimOS* simulates the computer hardware in enough detail to boot and run commercial operating systems. So far, SGI IRIX versions 5.3 (32-bit) and 6.4 (64-bit) as well as and Digital UNIX can be run. To reduce simulation overhead fast simulation techniques can be used to scan over the less interesting, time–consuming parts of a workload. Once more interesting sections of a workload's execution are reached, *SimOS* can be triggered to change to more detailed (and consequently slower) levels of simulation. The simulator provides a programmable TCL and Java interface to simulation events.

SimpleScalar tool set [BA97] is a processor simulation environment for the SimpleScalar architecture. The SimpleScalar architecture is a close derivate of the MIPS architecture [Pri95]. The tool set includes functional simulation, profiling, cache simulation, and out–of–order processor timing simulation. The functional simulation executes instructions serially without time accounting and assumes no caches. Furthermore, a functional simulator is included that produces manifold profile information. For cache simulation purpose, the tool set includes two cache simulators. The first implements a straightforward cache simulator for a two level memory hierarchy. The second is based on the *Cheetah* cache simulator. Both, however, do not take the latency of the caches into account. Finally, a simulator is provided which supports out–of–order issue and execution. This simulator also takes cache and memory latencies into account to measure the

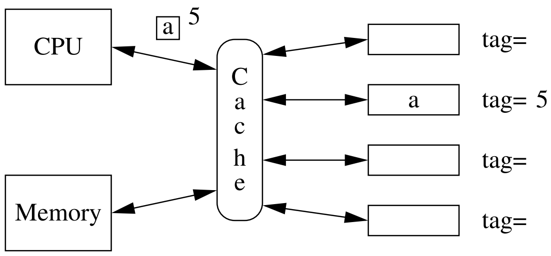

Figure 7.1: Snapshot of a cache animation

effect of cache organization upon the execution time of real programs.

7.3 Memory Hierarchy Visualization Techniques

Profiling tools and simulators produce data over time which is aggregated into summary measurements like miss rates, hit rates, or memory traffic information. With precise measurement or simulation cache misses can be classified as cold, capacity, or conflict misses. Furthermore, when misses are correlated to stall cycles even the total number of cycles waited for data to arrive from memory can be estimated. While this information is valuable to judge the quality of a program in respect to data locality this information is only of limited use for program optimizations.

Some tools, therefore, allow the events (cache misses, cache hits etc.) to be attributed to the source code lines. Thus, a programmer is able to identify statements which generate many misses. In some cases, this informations may be sufficient to guide data locality optimizations but in general the programmer will often need a more detailed understanding of the cache behavior of the program he wants to optimize.

Furthermore, the cache content is permanently changing during the execution of a program. Both techniques described above fail to capture the dynamic nature of caches and program traces. Visualization may provide a more intuitive and easy to understand representation of caches. Either the underlying physical structure of caches or the dynamic real–time behavior can be visualized; if necessary step by step. In the following, we will propose several cache visualization approaches which are useful for computer science education, software optimizations, and cache hardware design and optimization.

7.3.1 Cache Visualization for Education

In computer architecture lectures caches are usually introduced with block diagrams and verbal descriptions of the principle of locality and the cache hardware. Cache visualizations can extend block diagrams and provide a more intuitive representation of caches. Different cache designs such as different associativity can be demonstrated with step by step animations using moving data elements.

Figure 7.1 illustrates a snapshot of hypothetical cache visualization for educational purpose. The small cache allows the storage of four memory locations. Memory location 5 contains the data represented by “a”. The CPU requested this memory location and the cache just delivered the data to the CPU. Requests can be illustrated by moving elements annotated with the corresponding address and “h” for hits resp. “m” for misses, for example.

The necessary resolution and the low speed of the animation required for educational purpose, however, prohibits that this kind of visualization is used for software or hardware optimizations where possibly millions of memory references are performed.

7.3.2 Complete Program Run Visualization

For this kind of application the visualization of complete program runs is more appropriate. The cache behavior of the whole program is illustrated in a single picture and the human pattern perception ability is exploited to detect abnormal cache behavior, computation hot spots, or performance bottlenecks. The visualization should allow zooming so that the programmer can examine the part of interest in more detail. Furthermore, automatic correlation to the source code is desired. Ideally, the visualization tool should be able to spawn a debugger or a more detailed visualization at the point of execution which the user determines by means of the visualization. In the following we will propose several visualization techniques for complete program runs:

- Strip charts
- Colored address–memory reference plots
- Scatter plots

Strip Charts

Strip charts are a traditional approach to represent statistical data over a long period of time. They show the changing of a (single) value over time. In the context of cache visualization the value to be displayed on the y–axis can be for example the aggregated number of cache misses, cache hits, or memory traffic during a certain period of time. The period of time can be defined by a certain number of cycles or a certain number of memory references. The first possibility has the drawback that a high number of cache misses will introduce data stalls and therefore in turn reduce the number of misses in a given time (=cycle) interval. Thus, strip charts based on memory references like the one shown in Figure 7.2 seem to be more reasonably. Figure 7.2 represents the cache miss distribution of a hypothetical program over time. The program is featured by a periodic access behavior with a high number of cache misses at the beginning of a period. Furthermore, the program execution contains two repeated periods with different levels of cache misses in each period.

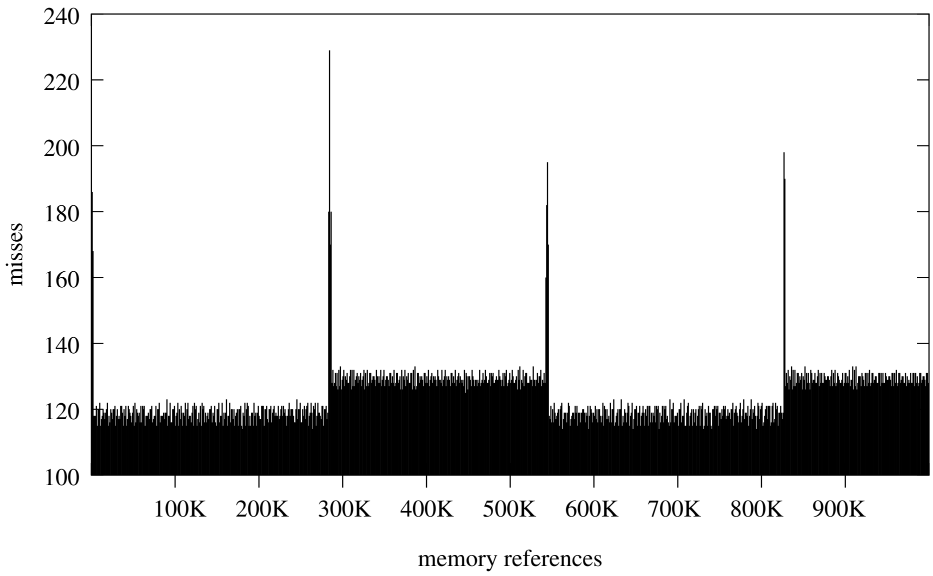

Figure 7.2: A strip chart sample showing cache misses distribution over time.

Zooming can be used to narrow the period of time displayed by the visualization. However, the misses represented by one pixel will then be aggregated over a smaller period of time so that the number of misses drawn at the y–axis will be reduced.

Strip charts can be used to identify hot spots in the computation. However, the chart is highly dependent on the resolution used on the x–axis. If too many memory references are mapped to one pixel repeatedly appearing memory bottlenecks may not be visible. For example, the repeated cache behavior of the program execution illustrated in Figure 7.2 will not be visible if the two different periods are mapped to a single point on the x–axis since the misses in the two periods will be aggregated to a single value. Thus, the aggregated miss rates shown in the strip chart will be almost constant over time.

Colored Address–Memory Reference Plots

The colored address–memory plot draws memory references versus the memory addresses of the references. Similar to strip charts memory references are drawn on the x–axis. The addresses of the memory references are spread over the y–axis. Again several memory references may be represented by one point on the x–axis, i.e. the plot may include vertical lines. A pixel (x, y) is plotted if one of the memory references to the memory address y in the memory reference interval x caused a cache miss. To represent additional information the pixels can be colored. For example, high numbers of cache misses can be colored black whereas small numbers of cache misses are represented by bright colors.

The address–memory plots can be used to identify abnormal cache behavior of programs such as a high number of conflict misses due to cross or self interference of arrays. Figure 7.3 shows the cache behavior of a hypothetical program with three typical cache

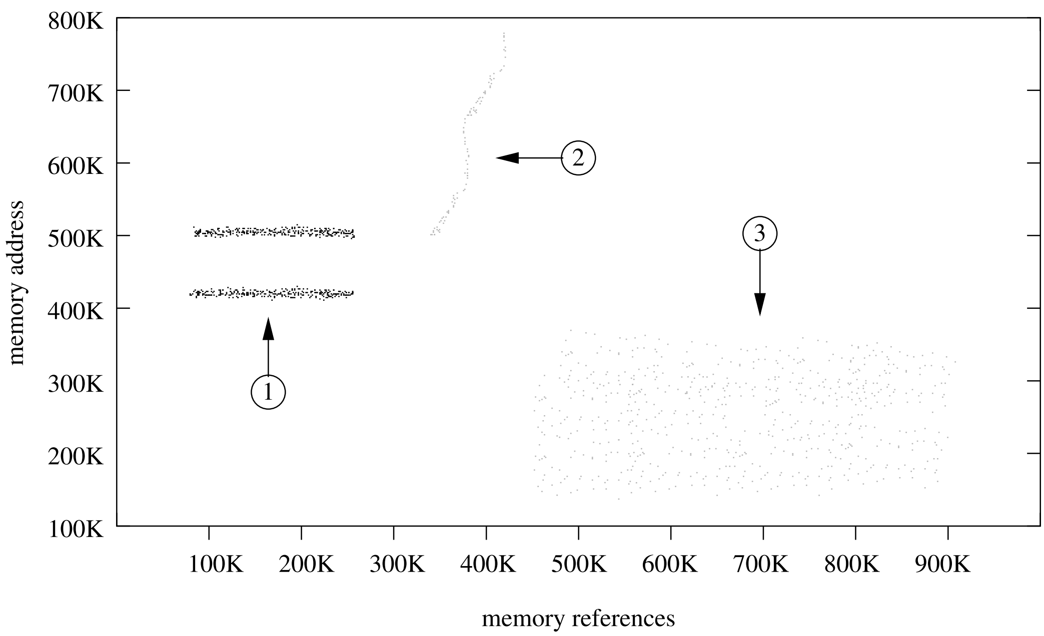

Figure 7.3: Address–memory reference plots

behaviors. In Area 1 the program repeatedly accesses two memory addresses which are causing a high number of conflict misses. The two memory areas are mapped to the same cache lines, replacing each other. Thus, severe conflict misses are represented by two dark horizontal lines located above each other. Furthermore, the visualization exposes a streaming behavior of the program in Area 2 and a random access pattern in Area 3.

The main disadvantage of the view is that if the interval which is represented by a point on the x–axis is small the plot will only be sparsely populated. In this case, the cache lines which are accessed by memory references could be displayed on the y–axis instead of their main memory addresses.

Scatter Plots

A scatter plot is a plot of the values of y versus the corresponding values of x. The variable y is usually the response variable and variable x is usually some variable we suspect may effect the response. A scatter plot reveals relationships or associations between the two variables. Such relationships manifest themselves by any non–random structure in the plot.

Scatter plots can be used to visualize the effect of memory references upon the cache. Therefore, each axis represents the address space of a program execution. For each data element evicted from the cache a pixel is plotted at the (x, y) coordinate with y being the memory location (address) which got evicted and x being the memory address which replaced the data.

In the case of memory address scatter plots a relationship is equivalent to two memory

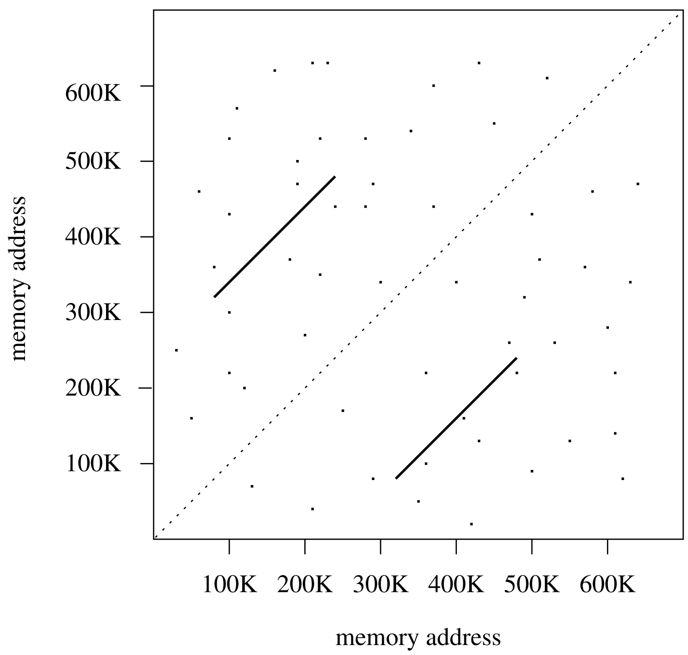

Figure 7.4: Memory Address Scatter Plot

regions which interfere in the cache. Thus, scatter plots reveal conflict misses by means of non–random patterns in the plot.

A hypothetical scatter plot of a program execution is shown in Figure 7.4. Besides other memory accesses the program alternately accesses two array. The first array is located from memory address 80K to 240K whereas the second is located from 320K to 480K. The arrays happen to be mapped to the same cache lines. Therefore, they cause a high number of cross interference misses which is pointed out by two diagonal lines. In our example, the data of other memory references got replaced as well but since the pattern for this memory references is random no relationship can be determined.

The drawback of scatter plots is that the correlation to the program flow is not represented in the visualization. Thus, bad data layouts can be identified but not bad sequences of the program execution.

7.3.3 Dynamic Cache Visualization

The visualization of complete program executions allows the identification of suspicious memory access patterns and memory bottlenecks. Once a pattern or bottleneck is identified the programmer may want to examine this part of the program in more detail. While zooming functionality for the program execution views can further assist the programmer in identifying the program part to focus on, all complete program execution visualization fail to capture the dynamic nature of the caches in enough detail.

For this purpose an animated representation of the cache or memory content is required. The representation should allow step by step execution, include navigation elements such as play, fast forward, rewind, stop etc. and maintain a source code relation.

Memory Content or Memory Reference Visualization

Visualization of memory contents in combination with the animation of misses is able to give a good overview of the dynamic cache behavior of a small part of the execution. The difficulty with memory content visualization, however, is that the amount of data to be visualized may be very large. If a program uses 1 Gbyte of memory approximately a 32000×32000 picture will be required if each memory location which is actually used is represented by one pixel. If some unused parts are visualized as well, even larger pictures will be required. A picture of that size is too large to keep the whole memory behavior in view. The display may be scaled down significantly if a single pixel is used to represent a memory block equal to the size of a cache line instead of one memory location. If this is still insufficient the size of the picture must be reduced further with the loose of information. A memory content display may be build up as follows.

Memory locations resp. memory blocks which are adjacent in memory are to be displayed as horizontally adjacent pixels whereas vertically adjacent pixels represent memory locations resp. memory blocks which are mapped to the same cache line. Thus, memory locations visualized in the the same column of the display cause conflicts in the cache. The pixels are colored to represent additional information. For example memory locations which recently suffered from many misses are represented by dark red pixels, memory location with a few misses are less dark, and memory locations which are not accessed at all or do not introduce misses are drawn with white pixels.

Cache Content Visualization

Low level information can be observed with a cache content visualization. The cache content and its evolution is visualized over time. Each cache line can be represented by a rectangular box or as a pixel if the cache is large. A click on a box resp. pixel may reveal additional information about the cache line such as the memory address currently cached by the line or a small access history. The cache lines may be colored to accommodate further information. For example, the cache lines can be colored so that cache lines belonging to different arrays have distinct colors. The cache lines may also be colored according to recent hit or miss rates. Thus, if for a cache line many misses occurred recently the cache line could be colored red. The color then fades over time when no additional misses happen. In general, the cache content view can be compared to a low level source code debugger.

7.3.4 MHVT: A Memory Hierarchy Visualization Tool

The Memory Hierarchy Visualization Tool (*MHVT*) is a design study for cache simulation and visualization tools. The tool is implemented in Java and allows the simulation and visualization of complete memory hierarchies. The simulation is so far based on trace files. A simulation based on source code, however, is planed. The simulator allows cache parameters such as cache type, cache size, cache lines size, associativity, replacement strategy etc. to be set for each cache level.

Based on the detailed cache simulation visualizations for each level of the memory hierarchy can be implemented. So far, the tool only includes the complete program run visualizations strip chart (for cache misses and cache hits) and the colored address–memory reference plot. All views allow zooming. A correlation of events to source code, however, is not possible since the simulation is based on trace files.

Besides the implemented complete program views the tool also supports dynamic cache visualization by means of an event notification and a program navigation system. The visualization tools can register for a variety of events such as cache misses, cache hits, etc. for any cache in the system. The navigation system which is also equipped with an event notification itself allows to control the program trace simulation. Currently, single step, play, fast forward, rewind, and jump to end actions are supported.

7.3.5 Existing Visualization Tools

Other existing cache visualization tools (*CVT*, *Thor*, and *CacheViz*) are described in the following.

CVT [vdDTGK97] is a cache visualization system which was developed with the intention to provide some sort of cache debugger. Thus, *CVT* allows the display of a single cache level and its evolution during a program execution. Cache lines are visualized as adjacent rectangular boxes. The boxes are colored during execution to represent the array which the cache line currently keeps data of. A click on a box reveals further information like miss rates for the specific cache line, a small history and the current cache line content. The visualized data is provided by a builtin cache simulator. The cache simulator allows to set the parameters cache size, cache line size, associativity, and the write policy. If this is not sufficient another simulator can be plugged into the *CVT*. Data references for the simulator can be provided by three different ways. First, loop nest in a specific source code format can be used. The loop nest may only contain array definitions at the beginning as well as loop statements and array references. The scope of programs presentable with the format is limited, however, may be sufficient to study inner kernels of numerical algorithms such as the red–black Gauss–Seidel algorithm. The advantage of this input format is that the array base addresses and parameters like block sizes can be easily modified. The other two input formats are trace based. The first is a source code trace which contains information about the referenced address, the corresponding array name and array base. The source code trace file is generated by adding calls to output routines to the source code of the studied program and running the modified program after recompiling. The second trace format is a regular trace format which can be generated by instrumentation tools like *Atom*. Both formats allow visualization of arbitrary program runs, the correlation to the source code, however, is limited.

The *Thor* visualization which is part of the *Rivet* system [BST$^+$00] presents detailed summaries of memory system utilization data collected by FlashPoint, a firmware memory profiler running on the FLASH multiprocessor. The information available on the home page[2] and in [BST+00], however, is to vague about *Thor* to present an adequate survey of the tool functionality besides the fact that it includes some cache and memory visualization.

Recently, the cache visualization tool *CacheViz* was released to the public [YBD01]. The focus of the tool is on presenting the cache access behavior of a complete program execution in a single picture. Within the picture the user, hopefully, will recognize suspicious patterns which result in poor performance. The visualizations include a density, a reuse distance, and a histogram view. The first two visualizations use one pixel per memory reference. Horizontally adjacent pixels represent consecutive memory references. The pixel line is horizontally wrapped. The visualizations use different pixel colorings to present further information. The density view colors the pixel white if the corresponding memory reference resulted in a cache hit, blue in case of a cold miss, green in case of a capacity miss, and red in case of a conflict miss. The reuse distance view colors pixels according to the reuse distance of the memory reference. The reuse distance is defined as the distinct number of memory references between two memory references to the same memory location. Memory references with very short reuse distance are displayed in blue within the visualization. With growing reuse distance the color of the pixels representing a memory reference is shifted towards red. Finally, a histogram view represents the distribution of reuse distances. The drawback of the memory reference plots is that the program execution is no longer linearly represented. Consequently, the correlation of a pixel to the corresponding execution time is difficult.

7.4 Program Transformation Tools

The ideal case for a programmer is that the compiler automatically applies data locality optimizations. Compiler systems which perform automatic parallelization of programs have been subject to a lot of research. See for example [HAA+96, KLS94, PGH+99, BCG+95] to mention some. Among them only the *SUIF* [AL93, AAL95, LLL01] and the *PARADIGM* [KCRB98, KCR+98a, KCR+98b] compiler system are able to perform data locality optimizations.

For complex programs, however, these compiler systems are still not able to apply data locality without user interaction. The reason for this may be that the series of transformations which leads to an improved locality is too complex or some existing data dependences prohibit the optimization. In some cases a compiler system might in fact assume a data dependence for a loop nest which prohibits optimizations although there is none since compilers have limited knowledge of the program behavior and data. Such situations frequently occur, for example, in languages like C or C++ which support pointer arithmetic.

[2] http://www-graphics.stanford.edu/projects/rivet/

If the transformation is not applied automatically the programmer needs to modify his program manually so that the locality is improved and the program semantic is maintained. The series of transformations required to be applied to the program to achieve a good data locality is often complex. Applying the transformation manually is difficult and requires a lot of testing and debugging. The task is complicated further by the fact that the behavior of the cache itself and the behavior of a program executed on a memory hierarchy is not common knowledge.

Consequently, the programmer requires at least some means to securely perform loop transformations. Unfortunately, the compiler community has not yet provided compiler directives for data locality optimizations like the OpenMP [DM98] directives for parallelization.

Interactive tools [KMT91, Che92, LDB$^+$99, Lia00] could be an alternative to complex compilers. They usually provide program transformations such as loop unrolling, loop interchange, loop fusion, etc. within a menu based user interface. The focus of these tools, however, is mainly on parallelization and the tools cannot be easily extended.

Furthermore, there are a series of tools [BBG$^+$94, BDE$^+$96, SD00] which provide an infrastructure to build source–to–source transformation tools. They mainly consist of parsers, representations of program statements as abstract syntax trees (AST), some manipulation routines for the AST, and unparsers. While these tools can be valuable for compiler writers, they are usually to complex for application programmers.

Finally, there are some xcode transformation tools based on pattern–matching available [HBR96, Keß96, KP93, BMQ98]. Transformations are described with a pattern matching language and a set of actions to be performed on matches. The main disadvantage of such systems is that patterns cannot be easily extended. Research on how to specify arbitrary pattern is still ongoing [BMQ98, Qui00, QP01].

7.5 Summary

In principle, programmers would like cache optimizations to be applied fully automatically to their programs. Although there are lots of efforts to automate cache optimizations, the optimization of complex programs is still out of scope for available compiler systems. Thus, the programmer faces the problem of performance analysis and code transformation for data locality which tend to be very complex.

Traditional performance analysis tools are only of limited use for the detection of memory performance bottlenecks. The reason for this is founded in the design principle of caches: caches are only visible for the microprocessors memory system but not for the software layer on top of the microprocessor. Thus, the functionality of performance tools is limited to indirect data locality measurements.

If the cache is not directly observable cache simulation can give some insight in the cache performance of programs. Unfortunately, cache simulation is often too slow and too unprecise to be of any value. The tendency to build more and more complex memory systems will make the simulation of caches either even more expensive or more inaccurate.

Fortunately, microprocessor manufacturers have realized the fact that caches must be visible for performance analysis tools. Therefore, they nowadays integrate hardware counters to measure a variety of hardware events including cache related events. However, the microprocessor developers typically do not provide ready–to–use tools to utilize the counters and third party tools for hardware counters are still rare or only available for a small number of platforms. Note, that there is some effort in providing a platform independent interface to hardware counters but the functionality of the tools based on top of the interface is still limited. Nevertheless, hardware counters so far provide the best means for data locality performance analysis.

All of the tools, however, fail to capture the dynamic nature inherently present in memory hierarchies. Cache visualization such as the visualizations proposed in this chapter could assist the programmer with program execution overviews or dynamic animations of the cache content. Tool writers, however, seem to have neglected cache visualizations so far. Thus, only a few very limited cache visualization tools are currently available.

To summarize, although there is at least some tool support for data locality optimizations, the support is still unsatisfying. The functionality of the tools is weak, portability of the tools among platforms is not guaranteed, and the automation of the optimization task is not to be expected in the near future. Based on the fact that the importance of good data locality for program performance is growing tool developers should focus more on tools for data locality optimizations.

Chapter 8

Conclusion

This thesis presents a detailed study of the runtime and memory behavior of multigrid methods and proposes and evaluates new optimization techniques to improve the data locality of multigrid methods on structured grids. This chapter concludes the thesis by giving a summary of the conducted work and by presenting a brief outlook on future work.

8.1 Summary

Multigrid algorithms belong to the most efficient methods for the solution of partial differential equations. On currently available computer systems, however, multigrid methods only achieve a small fraction of the theoretical peak performance. The increasing gap between microprocessor and main memory speed is the main cause for the bad performance. Nowadays, good data locality is crucial for the performance of programs on RISC microprocessors with deep memory hierarchies. As conventional multigrid methods have bad data locality properties, they will perform badly.

Starting from the observation that multigrid methods perform badly on computer systems with deep memory hierarchies, the runtime and memory behavior of multigrid methods has been systematically analyzed in this thesis. The results prove that the runtime of multigrid programs is dominated by delays implied by main memory accesses. In fact, microprocessors executing multigrid programs with large grid sizes are idle for over 50 per cent of all cycles, waiting for data to arrive from main memory. Multigrid methods repeatedly process large data structures, but the size of the data structures prevents the data to be cached. Thus, multigrid methods inherently possess good data locality, but the cache sizes of available or forecasted systems are too small to exploit it.

The study also reveals that besides the data locality issue the instruction mix involved with multigrid methods prevents that these kinds of algorithms exploit more than 50 per cent of the peak performance of typical microprocessors, even if a perfect memory system (with no latency and unlimited bandwidth) is assumed. The reason for this is that multigrid methods do not perform enough operations on the data once it is delivered to the CPU. Additional hardware resources within the microprocessor for executing load and store operations are required to overcome this limitation.

The core of the thesis is based on the detailed runtime and memory behavior study. In this thesis, new data locality optimizations for the smoother and inter–grid transfer operations of multigrid methods for structured grids are proposed and evaluated. The optimization techniques are able to significantly reduce the number of main memory references that occur in the smoother component which is typically the most time consuming part of a multigrid method. Furthermore, the techniques improve the utilization of higher levels of the memory hierarchy . Consequently, more than 80 per cent of the data is fetched from registers and the L1 cache even for large grid sizes. The locality optimizations are able to improve the performance of a red–black Gauss–Seidel code with constant coefficients on two–dimensional structured grids by factors of two to five for large grid sizes on many current computer systems.

The optimized smoother and inter–grid transfer routines have been integrated in the multigrid library DiMEPACK to demonstrate their applicability. The library provides an easy interface to several multigrid algorithms which can be used as building blocks for more complex algorithms. Thus, existing solvers in complex algorithms can be replaced by DiMEPACK routines and thereby significantly speed up their execution.

Finally, the role of data locality optimization tools is discussed based on the experience gained during the analysis of multigrid methods. Currently, the functionality and portability of existing performance analysis tools is limited in respect to data locality optimizations: profiling tools do not provide cache related information, cache simulation is too slow or too unprecise, hardware counter tools are not portable, and most of the program transformation tools are designed for parallelization and not adequate for data locality optimizations.

Cache visualization tools may overcome some problems involved with cache performance analysis, but these kinds of tools seem to be neglected by tool developers. In fact, only some research tools with limited functionality currently exist. Therefore, several new cache visualization techniques for complete program runs have been proposed in this thesis and implemented in the experimental visualization tool *MHVT*.

8.2 Applicability

The data locality optimization techniques described in this thesis have been integrated in the multigrid library DiMEPACK. DiMEPACK is able to solve problems with constant coefficients on two–dimensional structured grids. It has been demonstrated that DiMEPACK can be used to replace an existing solver for the Poisson equation in a chip placement code. However, the requirement that the coefficients are constant limits the applicability of DiMEPACK.

Although the description of the fundamental optimization techniques in this thesis focuses on optimization techniques for multigrid methods on structured grids with constant coefficients for two–dimensional problems, also issues involved with generalizations for variable coefficient and three–dimensional problems have been discussed. Based on the work described in this thesis, other researchers [Kow01, Pfä00, RT00] have demonstrated that the optimization techniques are able to accelerate the execution of more general methods as well. The inclusion of efficient solvers with variable coefficients into DiMEPACK is therefore just a coding issue.

The major drawback of the techniques is that their automated application is not yet ensured by appropriate tool or compiler support, respectively. Some issues involved with the automation have been discussed in this thesis by means of the fusion technique and by [RT00]. However, the integration of these techniques in compiler systems is still ongoing research.

8.3 Outlook

The work described in this thesis can be extended in three different research directions:

Firstly, the tool support for data locality performance analysis and code transformation can be investigated further. Data locality optimizations are rarely used to improve the performance of a program, as the knowledge of the cache structure, memory, and cache behavior of algorithms is still limited to experts. Cache visualization techniques are a promising approach to make knowledge about cache and memory behavior of algorithms accessible to a wider audience.

Secondly, based on this thesis new multigrid methods which have inherently good data locality could be developed. The new methods could be based on optimization techniques which, in contrast to the techniques described in this thesis, do not produce bitwise identical results but utilize completely new approaches such as patch–wise adaptive multigrid methods [Löt97, LR97].

Thirdly, the data locality optimization techniques described in this thesis can be extended to architectures with multiple microprocessors on a chip. Several approaches — often based on domain decomposition techniques — have already been investigated to parallelize multigrid methods. The techniques usually assume communication to happen via shared memory or message passing. However, the new upcoming architectures with several microprocessor cores on a single chip, such as implemented in the MAJC–5200 or the announced Power 4 microprocessor support communication through on–chip caches. Thus, data locality optimization techniques which improve the locality in such an environment will significantly improve performance.

Fourthly, it has to be demonstrated that the techniques described in this thesis are able to improve the data locality of other iterative methods beside multigrid methods as well.

Bibliography

[AAL95] J.M. Anderson, S.P. Amarasinghe, and M.S. Lam. Data and Computation Transformations for Multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 166–178, Santa Barbara, California, USA, July 1995.

[ABB+99] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorenson. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, Pennsylvania, USA, 3rd edition, 1999.

[ABD+97] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 1–14, St. Malo, France, October 1997.

[Adv99] Advanced Micro Devices, Inc., Sunnyvale, California, USA. *AMD Athlon Processor Technical Brief*, December 1999. Order number: 22054D.

[Adv00a] Advanced Micro Devices, Inc., Sunnyvale, California, USA. *AMD Athlon Processor and AMD Duron Processor with Full–Speed On–Die L2 Cache*, June 2000. Available as White Paper.

[Adv00b] Advanced Micro Devices, Inc., Sunnyvale, California, USA. *AMD Athlon Processor Architecture*, August 2000. Available as White Paper.

[AK84] R. Allen and K. Kennedy. Automatic Loop Interchange. In *Proceedings of the SIGPLAN Symposium on Compiler Constructs*, volume 19 of *SIGPLAN Notices*, pages 233–246, Montreal, Canada, June 1984.

[AK87] R. Allen and K. Kennedy. Automatic Translation of Fortran Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[AL93] J.M. Anderson and M.S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of the Conference* *on Programming Language Design and Implementation*, pages 112–125, Albuquerque, New Mexico, USA, June 1993.

[Amd67] G.M. Amdahl. Validity of the Single–Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, Atlantic City, New Jersey, USA, April 1967. AFIPS Press.

[ASW+93] S.G. Abraham, R.A. Sugumar, D. Windheiser, B.R. Rau, and R. Gupta. Predictability of Load/Store Instruction Latencies. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 139–152, December 1993.

[Ato96] Compaq Computer Corporation, Maynard, Massachusetts, USA. *Programmer's Guide, Digital UNIX Version 4.0*, March 1996. Order number: AA-PS30D-TE.

[BA97] D. Burger and T.M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, Computer Science Department, University of Wisconsin–Madision, Madison, Wisconsin, USA, June 1997.

[BACD97] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing Matrix Multiply using PHiPAC: A Portable, High–Performance, ANSI C Coding Methodology. In *Proceedings of the International Conference on Supercomputing*, July 1997.

[Ban76] U. Banerjee. Dependence Testing in Ordinary Programs. Master's thesis, Department of Computer Science, University of Illinois, Urbana–Champaign, Illinois, USA, November 1976.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publisher, 1988.

[Bas91] F. Baskett. Keynote Address. Held on the International Symposium on Shared Memory Multiprocessing, April 1991.

[BBG+94] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: An Object–Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. In *Proceedings of the 2nd Object Oriented Numerics Conference (OON–SKI)*, pages 122–136, Sunriver, Oregon, USA, April 1994.

[BCG+95] P. Banerjee, J.A. Chandy, M. Gupta, E.W. Hodges IV, J.G. Holm, A. Lain, D.J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed–Memory Multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.

[BDE+96] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel Programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[BDG+00] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

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

[Bel66] L.A. Belady. A Study of Replacement Algorithms for a Virtual–Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[BFS89] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but Effective Techniques for NUMA Memory Managment. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 19–31, Litchfield Park, Arizona, USA, December 1989.

[BG97] D. Burger and J.R. Goodman. Billion–Transistor Architectures – Guest Editor's Introduction. *IEEE Computer*, 30(9):46–49, September 1997.

[BGK95] D.C. Burger, J.R. Goodman, and A. Kägi. The Declining Effectiveness of Dynamic Caching for General–Porpuse Microprocessors. Technical Report CS TR-95-1261, Computer Sciences Department, University of Wisconsin–Madison, Madison, Wisconsin, USA, 1995.

[BGS94] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High–Performance Computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[BHM00] W.L. Briggs, V.E. Henson, and S.F. McCormick. *A Multigrid Tutorial*. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, USA, second edition, 2000.

[BL96] T. Ball and J.R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 46–57, Paris, France, December 1996.

[BM00] R. Berrendorf and B. Mohr. PCL – The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors. Technical report, Reserch Centre Juelich GmbH, Juelich, Germany, September 2000. http://www.fz-juelich.de/zam/PCL/.

[BMQ98] F. Bodin, Y. Mevel, and R. Quiniou. A User Level Program Transformation Tool. In *Proceedings of the International Conference on Supercomputing*, pages 180–187, Melbourne, Australia, July 1998.

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

[BST+00] R. Bosch, C. Stolte, D. Tang, J. Gerth, M. Rosenblum, and P. Hanrahan. Rivet: A Flexible Environment for Computer Systems Visualization. *ACM SIGGRAPH Newsletter: Computer Graphics*, 34(1), February 2000.

[Cas99] B. Case. Sun Makes MAJC With Mirrors. *Microprocessor Report*, 13(14), October 1999.

[CB94] T.-F. Chen and J.-L. Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 223–232, April 1994.

[CB95] T.-F. Chen and J.-L. Baer. Effective Hardware Based Data Prefetching for High–Performance Processors. *IEEE Transactions on Computers*, 44(5):609–623, 1995.

[Che92] D.Y. Cheng. Evaluation of Forge: An Interactive Parallelization Tool. In *Proceedings of the Twenty–Fifth Hawaii International Conference on System Sciences*, volume 2, pages 287–297, January 1992.

[CKP91] D. Callahan, K. Kennedy, and A.K. Porterfield. Software Prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[CL95] M. Cierniak and W. Li. Unifying Data and Control Transformations for Distributed Shared–memory Machines. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 205–217, La Jolla, California, USA, June 1995.

[Com97] Compaq Computer Corporation. *Digital Semiconductor 21164 Alpha Microprocessor Hardware Reference Manual*, February 1997. Order number: EC-QP99B-TE.

[Com99] Compaq Computer Corporation. *Alpha 21264 Microprocessor Data Sheet*, February 1999. Order number: EC-R4CFA-TE.

[Com01] Compaq Computer Corporation. *Hiprof – Hierarchical Instruction Profiler*, 2001. Tru64 Unix Manual Page V4.0.

[Dar99] A. Darte. On the Complexity of Loop Fusion. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, pages 149–157, Newport Beach, California, USA, October 1999.

[DE73] G. Danzig and B.C. Eaves. Fourier–Motzkin Elimination and Its Dual. *Journal of Combinatorial Theory*, 14:288–297, 1973.

[DHH+00a] C.C. Douglas, G. Haase, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß. Portable Memory Hierarchy Techniques For PDE Solvers: Part I. *Siam News*, 33(5), June 2000.

[DHH+00b] C.C. Douglas, G. Haase, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß. Portable Memory Hierarchy Techniques For PDE Solvers: Part II. *Siam News*, 33(6), July 2000.

[DHI+00] C.C. Douglas, J. Hu, M. Iskandarani, M. Kowarschik, U. Rüde, and C. Weiß. Maximizing Cache Memory Usage for Multigrid Algorithms. In Z. Chen, R.E. Ewing, and Z.-C. Shi, editors, *Numerical Treatment of Multiphase Flows in Porous Media. Proceedings of the International Workshop Held at Beijing, China, 2-6 August, 1999*, volume 21 of *Lecture Notes in Physics*, pages 124–137. Springer, August 2000.

[DHK+00a] C.C. Douglas, J. Hu, W. Karl, M. Kowarschik, U. Rüde, and C. Weiß. Fixed and Adaptive Cache Aware Algorithms for Multigrid Methods. In E. Dick, K. Riemslagh, and J. Vierendeels, editors, *Multigrid Methods VI. Proceedings of the Sixth European Multigrid Conference held in Gent, Belgium, September 27-30, 1999*, volume 14 of *Lecture Notes in Computer Science and Engineering*. Springer, July 2000.

[DHK+00b] C.C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transaction on Numerical Analysis*, 10:21–40, February 2000.

[Die99a] K. Diefendorff. Hal Makes Sparcs Fly. *Microprocessor Report*, 13(15), November 1999.

[Die99b] K. Diefendorff. Power4 Focuses on Memory Bandwidth. *Microprocessor Report*, 13(13), October 1999.

[DM98] L. Dagum and R. Menon. OpenMP: An Industry–Standard API for Shared–Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

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

[EEL+97] S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, R.L. Stamm, and D.M. Tullsen. Simultaneous Multithreading: A Platform for Next–Generation Processors. *IEEE Micro*, 17(5):12–19, October 1997.

[EJ98] H. Eisenmann and F.M. Johannes. Generic Global Placement and Floorplanning. In *Proceedings of the 35th Annual International ACM/IEEE Design Automation Conference (DAC)*, pages 269–274, San Francisco, California, USA, June 1998.

[ES95] A. Eustace and A. Srivastava. ATOM: A Flexible Interface for Building High Performance Program Analysis Tools. In *Proceedings of the Winter 1995 USENIX Technical Conference on UNIX and Advanced Computing Systems*, pages 303–314, January 1995.

[FJ97] M. Frigo and S.G. Johnson. The Fastest Fourier Transform in the West. Technical Report MIT-LCS-TR-728, Massachusetts Institute of Technology, September 1997.

[FS98] J. Fenlason and R. Stallman. *GNU Gprof*. Free Software Foundation, Inc., Boston, Massachusetts, USA, 1998.

[FST91] J. Ferrante, V. Sarkar, and W. Trash. On Estimating and Enhancing Cache Effectiveness. In U. Banerjee, editor, *Fourth International Workshop on Languages and Compilers for Parallel Computing*. Springer, August 1991.

[GJG88] D. Gannon, W. Jalby, and K. Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988.

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

[GKT91] G. Goff, K. Kennedy, and C.-W. Tseng. Practical Dependence Testing. In *Proceedings of the SIGPLAN'91 Symposium on Programming Language Design and Implementation*, volume 26 of *SIGPLAN Notices*, pages 15–29, Toronto, Canada, June 1991.

[GL83] G.H. Golub and C.F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press and North Oxford Academic, 1983.

[GL01] D. Genius and S. Lelait. A Case for Array Merging in Memory Hierarchies. In *Proceedings of the 9th Workshop on Compilers for Parallel Computers (CPC'01)*, Edinburgh, Scotland, June 2001.

[Gla00] P.N. Glaskowsky. Pentium 4 (Partially) Previewed. *Microprocessor Report*, 14(35), August 2000.

[GMM99] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, July 1999.

[GMM00] S. Ghosh, M. Martonosi, and S. Malik. Automated Cache Optimizations Using CME Driven Diagnosis. In *Proceedings of the International Conference on Supercomputing*, pages 316–326, Santa Fe, New Mexico, USA, May 2000.

[Gwe96a] L. Gwennap. Digital 21264 Sets New Standard. *Microprocessor Report*, 10(14), October 1996.

[Gwe96b] L. Gwennap. HP Pumps Up PA–8x00 Family. *Microprocessor Report*, 10(14), October 1996.

[Gwe98a] L. Gwennap. Alpha 21364 to Ease Memory Bottleneck. *Microprocessor Report*, 12(14), October 1998.

[Gwe98b] L. Gwennap. Most Significant Bits: HP Extends PA–RISC Plans. *Microprocessor Report*, 12(15), November 1998.

[Gwe99a] L. Gwennap. Coppermine Outruns Athlon. *Microprocessor Report*, 13(14), October 1999.

[Gwe99b] L. Gwennap. Most Significant Bits: Alpha 21464 Targets 1.7 GHz in 2003. *Microprocessor Report*, pages 13–16, November 1999.

[Gwe99c] L. Gwennap. Most Significant Bits: PA–8600 in Early 2000. *Microprocessor Report*, 13(4), March 1999.

[HAA+96] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.-W. Liao, E. Bugnion, and M.S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler. *IEEE Computer*, 29(12):84–89, December 1996.

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

[Hac93] W. Hackbusch. *Iterative Solution of Large Sparse Systems of Equations*, volume 95 of *Applied Mathematical Sciences*. Springer, 1993.

[HBR96] J.R. Hagemeister, S. Bhansali, and C.S. Raghavendra. Implementation of a Pattern–Matching Approach for Identifying Algorithmic Concepts in Scientific Fortran Programs. In *Proceedings of the 1996 International Conference on High–Performance Computing (HiPC '96)*, pages 209–214, December 1996.

[Hil87] M.D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, Computer Science Division (EECS), University of California, Berkeley, California, USA, November 1987.

[HL99] T. Horel and G. Lauterbach. UltraSPARC–III: Designing Third–Generation 64–Bit Performance. *IEEE Micro*, 9(3), May 1999.

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

[HS89] M.D. Hill and A.J. Smith. Evaluating Associativity in CPU Caches. *IEEE Transaction on Computers*, 38(12):1612–1630, December 1989.

[Hu00] J. Hu. *Cache–Based Multigrid on Unstructured Grids in Two and Three Dimensions*. PhD thesis, Department of Methematics, University of Kentucky, Lexington, Kentucky, USA, 2000.

[Hun95] D. Hunt. Advanced Performance Features of the 64–bit PA–8000. In *Proceedings of COMPCON'95*, pages 123–128, March 1995.

[Int00a] Intel Corporation. *Itanium Processor Microarchitecture Reference*, August 2000. Document Number: 245473–002.

[Int00b] Intel Corporation. *Pentium III Processor for the PGA370 Socket at 500 MHz to 1 GHz*, October 2000. Order number: 245264-007.

[Int00c] Intel Corporation. *Pentium III Processor for the SC242 at 450 MHz to 1.13 GHz*, July 2000. Order number: 244452-008.

[Int01] Intel Corporation. *Intel Pentium 4 Processor in the 423–pin Package at 1.30 GHz, 1.40 GHz, and 1.50 GHz*, January 2001. Order number: 249198-002.

[ITR00] International Technology Roadmap for Semiconductors. http://public.itrs.net, 2000.

[JE95] T. Jeremiassen and S. Eggers. Reducing False Sharing on Shared Memory Multiprocessors Through Compile Time Data Transformation. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, Santa Barbara, California, USA, July 1995.

[Jou90] N. Jouppi. Improving Direct–Mapped Cache Performance by the Addition of a Small Fully–Associative Cache and Prefetch Buffers. In *Proceddings of the 17th International Symposium on Computer Architecture*, pages 388–397, Barcelona, Spain, June 1990.

[JT93] Y. Jegou and O. Temam. Speculative Prefetching. In *Proceddings of the International Conference on Supercomputing*, pages 57–66, Tokyo, Japan, July 1993.

[KAP97] I. Kodukula, N. Ahmed, and K. Pingali. Data–Centric Multi-Level Blocking. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 346–357, Las Vegas, Nevada, USA, June 1997.

[Kar93] W. Karl. *Parallele Prozessorarchitekturen*. Number 93 in Informatik. BI Wissenschaftsverlag, 1993.

[KCR+98a] M. Kandemir, A. Choudhary, J. Ramanujam, N. Shenoy, and P. Banerjee. A Matrix–Based Approach to the Global Locality Optimization Problem. In *Proceedings of Parallel Architectures and Compilation Techniques (PACT-98)*, Paris, France, October 1998.

[KCR+98b] M. Kandemir, A. Choudhary, J. Ramanujam, N. Shenoy, and P. Banerjee. Enhancing Spatial Locality Using Data Layout Optimizations. In D. Pritchard and J. Reeve, editors, *Proceedings of the European Conference on Parallel Processing (Euro–Par'98)*, volume 1470 of *Lecture Notes in Computer Science*, pages 422–434, Southampton, England, September 1998. Springer.

[KCRB98] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving Locality Using Loop and Data Transformations in an Integrated Framework. In *Proceedings of the 31st ACM/IEEE International Symposium on Micro–Architecture (MICRO–31)*, pages 285–297, Dallas, Texas, USA, December 1998.

[Keß96] C.W. Keßler. Pattern–Driven Automatic Parallelization. *Scientific Programming*, 5:251–274, 1996.

[KKRW01] W. Karl, M. Kowarschik, U. Rüde, and C. Weiß. DiMEPACK — A Cache–Aware Multigrid Library: User Manual. Technical Report 01–1, Lehrstuhl für Informatik 10 (Systemsimulation), University of Erlangen–Nuremberg, Erlangen, Germany, 2001. *www10.informatik.uni-erlangen.de*.

[KL91] A.C. Klaiber and H.M. Lewy. Architecture for Software–controlled Data Prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 43–63, Toronto, Canada, May 1991.

[KLS94] R.H. Kuhn, B. Leasure, and S.M. Shab. The KAP Parallelizer for DEC Fortran and DEC C Programs. *Digital Technical Journal*, 6(3):57–77, 1994.

[KMT91] K. Kennedy, K.S. McKinley, and C.-W. Tseng. Analysis and Transformation in the ParaScope Editor. In *Proceedings of the International Conference on Supercomputing*, pages 433–447, Cologne, Germany, June 1991.

[Kog81] P.M. Kogge. *The Architecture of Pipelined Computer*. McGraw–Hill, New York, USA, 1981.

[Kow01] M. Kowarschik. Enhancing the Cache Performance of Multigrid Codes on Structured Grids. Presented at the Copper Mountain Conference on Iterative Methods, Copper Mountain, Colorado, USA, April 2001.

[KP93] C.W. Keßler and W.J. Paul. Automatic Parallelization by Pattern–Matching. In J. Volkert, editor, *Parallel Computing: Proceedings of the Second International ACPC Conference*, volume 734 of *Lecture Notes in Computer Science*, Gmunden, Austria, October 1993. Springer.

[KPP+97] C.E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable Processors in the Billion–transistor Era: IRAM. *IEEE Computer*, 30(9):75–78, September 1997.

[Kre00a] K. Krewell. HP Extends PA–RISC with 8700. *Microprocessor Report*, 14(21), May 2000.

[Kre00b] K. Krewell. IBM's Power4 Unveiling Continues. *Microprocessor Report*, 14(47), November 2000.

[Kre00c] K. Krewell. Sun Shines on UltraSPARC III. *Microprocessor Report*, October 2000.

[Kro81] D. Kroft. Lockup–Free Instruction Fetch/Prefetch Cache Organisation. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, May 1981.

[KRWK00] M. Kowarschik, U. Rüde, C. Weiß, and W. Karl. Cache-Aware Multigrid Methods for Solving Poisson's Equation in Two Dimensions. *Computing*, 64(4):381–399, 2000.

[KW01] M. Kowarschik and C. Weiß. DiMEPACK — A Cache–Optimized Multigrid Library. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001)*, volume I, pages 425–430, Las Vegas, Nevada, USA, June 2001. CSREA, CSREA Press.

[LDB+99] S.-W. Liao, A. Diwan, R.P. Bosch, A. Ghuloum, and M.S. Lam. SUIF Explorer: An Interactive and Interprocedural Parallelizer. In *Proceedings of* *the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'99)*, pages 37–48, May 1999.

[LGH94] J. Laudron, A. Gupta, and M. Horowitz. Interleaving: A Multithreading Technique Targeting Multiprocessor and Workstations. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, October 1994.

[LH97] G. Lesartre and D. Hunt. PA–8500: The Continuing Evolution of the PA–8000 Family. In *Proceedings of COMPCOM'97*, March 1997.

[Lia00] S.-W. Liao. *SUIF Explorer: An Interactive and Interprocedural Parallelizer*. PhD thesis, Department of Computer Science, Stanford University, Stanford, California, USA, August 2000.

[LL97] A. LaMarca and R.E. Ladner. The Influence of Cache on the Performance of Sorting. In *Proceedings of the 8th Annual ACM SIAM Symposium on Discreate Algorithms*, pages 370–379, New Orleans, Louisiana, USA, January 1997.

[LLL01] A.W. Lim, S.-W. Liao, and M.S. Lam. Blocking and Array Contraction Across Arbitrarily Nested Loops Using Affine Partitioning. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 103–112, Snowbird, Utah, USA, June 2001.

[Löt97] H. Lötzbeyer. Parallele adaptive Mehrgitterverfahren. Diplomarbeit, Institut für Informatik, Technische Universität München, 1997.

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

[LRW91] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 63–74, Palo Alto, California, USA, April 1991.

[LS95] J.R. Larus and E. Schnarr. EEL: Machine–Independent Executable Editing. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 291–300, La Jolla, California, USA, June 1995.

[LS97] M.H. Lipasti and J.P. Shen. Superspeculative Microarchitecture for Beyond AD 2000. *IEEE Computer*, 30(9):59–66, September 1997.

[LW94] A. Lebeck and D. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer*, 27(10):15–26, October 1994.

[McC95] J.D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, December 1995.

[MHL91] D.E. Maydan, J.L. Hennessy, and M.S. Lam. An Efficient Method for Exact Dependence Analysis. In *Proceedings of the SIGPLAN'91 Symposium on Programming Language Design and Implementation*, volume 26 of *SIGPLAN Notices*, pages 1–14, Toronto, Canada, June 1991.

[Mic00] Chart Watch: Workstation Processor. Microprocessor Report. MicroDesign Resources, Sunnyvale, CA, USA, December 2000.

[MLG92] T.C. Mowry, M.S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[Mow94] T.C. Mowry. *Tolerating Latency Through Software–Controlled Data Prefetching*. PhD thesis, Computer Systems Laboratory, Stanford University, March 1994.

[Muc97] S.S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, San Francisco, California, USA, 1997.

[NO97] B.A. Nayfeh and K. Olukotun. A Single–chip Multiprocessor. *IEEE Computer*, 30(9):79–85, September 1997.

[PDM+98] M. Papermeister, R. Dinkjian, M. Mayfield, P. Lenk, B. Ciarfella, F. O'Connell, and R. DuPont. Next–Generation 64–Bit PowerPC Processor Design. *Micro News, IBM Microelectronics*, 4(4), 1998.

[Pfä00] H. Pfänder. Cache–optimierte Mehrgitterverfahren mit variablen Koeffizienten auf strukturierten Gittern. Master's thesis, IMMD10, Department of Computer Science, University of Erlangen–Nuremberg, Germany, Erlangen, Germany, December 2000.

[PGH+99] C.D. Polychronopoulos, M.B. Girkar, M.R. Haghighat, C.L. Lee, B.P. Leung, and D.A. Schouten. The Structure of Parafrase–2: An Advanced Parallelizing Compiler for C and Fortran. In *Languages and Compilers for Parallel Computing*, pages 423–453. MIT Press, 1999.

[PK94] S. Palacharla and R.E. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, April 1994.

[Por89] A.K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Department of Computer Science, Rice University, Houston, Texas, USA, May 1989.

[PPE+97] Y.N. Patt, S.J. Patel, M. Evers, D.H. Friendly, and J. Stark. One Billion Transistors, One Uniprocessor, One Chip. *IEEE Computer*, 30(9):51–57, September 1997.

[Pra00] D. Prairie. *AMD Athlon Processor Floating Point Capability*. Advanced Micro Devices, Inc., Sunnyvale, California, USA, August 2000. Published as White Paper.

[Pri95] C. Price. *MIPS IV Instruction Set, Revision 3.1*. MIPS Technologies, Inc., Mountain View, California, USA, January 1995.

[PW92] W. Pugh and D. Wonnacott. Eliminating False Data Dependences Using the Omega Test. In *Proceedings of the SIGPLAN'92 Symposium on Programming Language Design and Implementation*, volume 27 of *SIGPLAN Notices*, pages 140–151, San Francisco, California, USA, July 1992.

[QP01] D. Quinlan and B. Philip. ROSETTA: The Compile–Time Recognition Of Object–Oriented Library Abstractions. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001)*, Las Vegas, Nevada, USA, June 2001.

[Qui00] D. Quinlan. ROSE: Compiler Support for Object–Oriented Frameworks. *Parallel Processing Letter*, 10(2,3):215–226, 2000.

[RD96] J.A. Rivers and E.S. Davidson. Reducing Conflicts in Direct–Mapped Caches with a Temporality–Based Design. In *Proceedings of the 1996 International Conference on Parallel Processing*, volume 1, pages 154–163, Bloomingdale, Illinois, USA, August 1996.

[RDC+94] E. Rashid, E. Delano, K. Chan, M. Buckley, J. Zheng, F. Schumacher, G. Kurpanek, J. Shelton, T. Alexander, N. Noordeen, M. Ludwig, A. Scherer, C. Amir, D. Cheung, P. Sabada, R. Rajamani, N. Fiduccia, B. Ches, K. Eshghi, F. Eatock, D. Renfrow, J. Keller, P. Ilgenfritz, I. Krashinsky, D. Weatherspoon, S. Ranade, D. Goldberg, and W. Bryg. A CMOS RISC CPU with On–Chip Parallel Cache. In *Proceedings of IEEE International Solid–State Circuits Conference (ISSCC'94)*, pages 210–211, San Francisco, California, USA, February 1994. IEEE, New York, USA.

[RHWG95] M. Rosenblum, S.A. Herrod, E. Witchel, and A. Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology: Systems & Applications*, 4(3):34–43, 1995.

[RT98a] G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, Montreal, Canada, June 1998.

[RT98b] G. Rivera and C.-W. Tseng. Eliminating Conflict Misses for High Performance Architectures. In *Proceedings of the International Conference on Supercomputing*, Melbourne, Australia, July 1998.

[RT00] G. Rivera and C.-W. Tseng. Tiling Optimizations for 3D Scientific Computation. In *Proceedings of the ACM/IEEE SC00 Conference*, Dallas, Texas, USA, November 2000.

[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.

[Rüd98] U. Rüde. Technological Trends and their Impact on the Future of Supercomputing. In H.-J. Bungartz, F. Durst, and C. Zenger, editors, *High Performance Scientific and Engineering Computing, Proceedings of the International FORTWIHR Conference on HPSEC*, volume 8 of *Lecture Notes in Computer Science and Engineering*, pages 459–471. Springer, March 1998.

[SA93] R.A. Sugumar and S.G. Abraham. Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurements and Modeling of Computer Systems*, pages 24–35, Santa Clara, California, USA, May 1993.

[SBK+97] A.P. Scott, K.P. Burkhart, A. Kumar, R.M. Blumberg, and G.L. Ranson. Four–Way Superscalar PA–RISC Processor. *Hewlett–Packard Journal*, 48, August 1997.

[SC01] S. Sellappa and S. Chatterjee. Cache–Efficient Multigrid Algorithms. In *Proceedings of the 2001 International Conference on Computational Science (ICCS 2001)*, volume 2073 and 2074 of *Lecture Notes in Computer Science*, pages 107–116, San Francisco, California, USA, May 2001. Springer.

[SD00] G.-A. Silber and A. Darte. The Nestor Library: A Tool for Implementing Fortran Source to Source Transformations. In *Proceedings of the Eighth International Workshop on Compiler for Parallel Computers (CPC 2000)*, pages 327–335, Aussois, France, January 2000.

[SF91] G. Sohi and M. Franklin. High–Performance Data Memory Systems for Superscalar Processors. In *Proceedings of the Fourth Symposium on*

Archictural Support for Programming Languages and Operating Systems, pages 53–62, April 1991.

[SL99] Y. Song and Z. Li. New Tiling Techniques to Improve Cache Temporal Locality. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, pages 215–228, Atlanta, Georgia, USA, May 1999.

[Smi82] A.J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[Son97a] P. Song. IBM's Power3 to Replace P2SC. *Microprocessor Report*, 11(15), November 1997.

[Son97b] P. Song. UltraSparc–3 Aims at MP Servers. *Microprocessor Report*, 11(14), October 1997.

[SR97] L. Stals and U. Rüde. Techniques for Improving the Data Locality of Iterative Methods. Technical Report MRR97-038, School of Mathematical Science, Australian National University, October 1997.

[SRWH97] L. Stals, U. Rüde, C. Weiß, and H. Hellwagner. Data Local Iterative Methods for the Efficient Solution of Partial Differential Equations. In J. Noye, M. Teubner, and A. Gill, editors, *Proceedings of the The Eighth Biennial Computational Techniques and Applications Conference: CTAC97*, pages 655–662, Adelaide, Australia, September 1997.

[Sti00] A. Stiller. Bei Lichte betrachtet: Die Architekturen des Pentium 4 im Vergleich zu Pentium III und Athlon. *C't Magazin für Computer Technik*, (24):134–141, November 2000.

[Sud00] S. Sudharsanana. MAJC–5200: A High Performance Microprocessor for Multimedia Computing. Technical report, Sun Microsystems, 2000.

[Sun97] Sun Microsystems. *UltraSPARC–II: Second Generation SPARC v9 64–Bit Microprocessor With VIS*, July 1997. Order number: 802-7430-05.

[Sun99] Sun Microsystems. *MAJC Architectural Tutorial*, 1999.

[SV97] J.E. Smith and S. Vajapeyam. Trace processors: Moving to Fourth–Generation Microarchitectures. *IEEE Computer*, 30(9):68–74, September 1997.

[TD96] E.S. Tam and E.S. Davidson. Early Design Cycle Timing Simulation of Caches. Technical Report CSE-TR-317-96, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan, USA, November 1996.

[Tem98] O. Temam. Investigating Optimal Local Memory Performance. In *Proceedings ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, San Diego, California, USA, October 1998.

[TFJ94] O. Temam, C. Fricker, and W. Jalby. Cache Interference Phenomenon. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modelling Computer Systems*, pages 261–271, Santa Clara, California, USA, May 1994.

[TGJ93] O. Temam, E. Granston, and W. Jalby. To Copy or Not to Copy: A Compile–Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts. In *Proceedings of the ACM/IEEE SC93 Conference*, Portland, Oregon, USA, November 1993.

[TLH90] J. Torrellas, M. Lam, and J. Hennessy. Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume 2, pages 266–270, Pennsylvania, USA, August 1990.

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

[Tow76] R.A. Towl. *Control and Data Dependence for Program Transformations*. PhD thesis, Department of Computer Science, University of Illinois, Urban–Champaign, Illinois, USA, March 1976.

[TRTD98] E.S. Tam, J.A. Rivers, G.S. Tyson, and E.S. Davidson. mlcache: A Flexible Multi–Lateral Cache Simulator. In *Proceedings of the 6th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '98)*, pages 19–26, Montreal, Canada, July 1998.

[vdDTGK97] E. van der Deijl, O. Temam, E. Granston, and G. Kanbier. The Cache Visualization Tool. *IEEE Computer*, 30(7):71–78, July 1997.

[VL00] S.P. Vanderwiel and D.J. Lilja. Data Prefetching Mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.

[Wal88] D.R. Wall. Dependence Among Multi–Dimensional Array References. In *Proceddings of the International Conference on Supercomputing*, pages 418–428, St. Malo, France, July 1988.

[WD98] R.C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proceddings of the International Conference on Supercomputing*, Orlando, Florida, USA, November 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, USA, November 1999.

[WL91] M.E. Wolf and M.S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the SIGPLAN'91 Symposium on Programming Language Design and Implementation*, volume 26 of *SIGPLAN Notices*, pages 33–44, Toronto, Canada, June 1991.

[WM95] W.A. Wulf and S.A. McKee. Hitting the Memory Wall: Implication of the Obvious. *ACM Computer Architecture News*, 23(1):20–24, March 1995.

[Wol89a] M.J. Wolfe. More Iteration Space Tiling. In *Proceddings of the International Conference on Supercomputing*, pages 655–664, Reno, Nevada, USA, November 1989.

[Wol89b] M.J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Massachusetts, USA, 1989.

[Wol92] M.E. Wolfe. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Computer Systems Laboratory, Stanford University, Stanford, California, USA, August 1992.

[Wol96] M.R. Wolfe. *High–Perfomance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, California, USA, 1996.

[WT90] M.R. Wolfe and C.-W. Tseng. The Power Test For Data Dependence. Technical Report CS/E 90-015, Department of Computer Science and Engineering, Oregon Graduate Institute, Beaverton, Oregon, USA, August 1990.

[XZK00] L. Xiao, X. Zhang, and S.A. Kubricht. Improving Memory Performance of Sorting Algorithms. *ACM Journal of Experimental Algorithmics*, 5(3):1–22, 2000.

[YBD01] Y. Yijun, K. Beyls, and E.H. D'Hollander. Visualizing the Impact of the Cache on Program Execution. In *Proceedings of the Fifth International Conference on Information Visualisation*, pages 336–341, London, England, July 2001.

Index