Andreas Grävinghoff

On the Realization of Fine-Grained Multithreading in Software

Für die Seerosenprinzessin

"After a rare speech at the National Center for Atmospheric Research in Boulder, Colorado, in 1976, programmers in the audience had suddenly fallen silent when Cray offered to answer questions. He stood there for several minutes, waiting for their queries, but none came. When he left, the head of NCAR's computing division chided the programmers. 'Why didn't someone raise a hand ?' After a tense moment, one programmer replied, 'How do you talk to God?' "

from The SUPERMEN

Acknowledgments

I am grateful to Prof. Keller for our successful cooperation during the last five years, especially for providing the time to complete this work by freeing me from other tasks. In addition, this work would not have been possible without the necessary equipment, e.g. the Compaq workstation that was for development and evaluation of emulated multithreading. I am grateful to Prof. Ungerer from the Universität Augsburg for reviewing this thesis and our interesting discussions during the course of this work.

I thank the following people and organizations for their support during the course of this work: the HLRS in Stuttgart and the NIC in Jülich for granting access to their Cray T3E computer systems - software development and evaluation would have been impossible without access to these machines. The people from the Gastdozentenhaus "Heinrich Hertz" at the Universität Karlsruhe for guarding me while writing the first draft. Peter Bach and Michael Bosch for providing useful comments on an early draft - I am really looking forward to working with them again at ETAS GmbH in Stuttgart. Matthias Müller from the Universität Karlsruhe for interesting discussions about the E-registers in the Cray T3E. The Universitat Politecnica de Catalunya in Spain, namely Ernest Artiaga, for providing an implementation of the PARMACS macros based on POSIX threads that was used during the evaluation of emulated multithreading on the Compaq workstation. The Informatikrechnerbereich in Hagen, namely Enno deVries, for providing the DECcampus software used during software development. My colleagues from the chair of Prof. Schiffmann for the good teamwork even in times of stress.

My heartfelt thanks go to my family, Lutz, Ulrike and Tina Grävinghoff for their support and encouragement during all these years. Last but not least, I wish to thank my fiancée Kerstin Bernhardt for her outstanding support even while pursuing her own PhD in psychology.

Contents

List of Figures

List of Tables

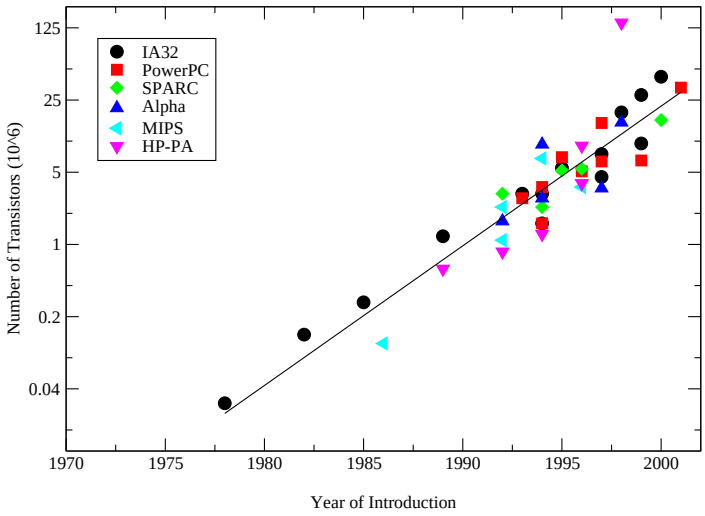
1. Introduction

This work deals with the design, implementation and evaluation of a multithreading system that enables fine-grained context switches without hardware support. The current chapter explains the rationale behind such a system and starts with an analysis of current trends in computer systems: Trends in sequential computing, i.e. single-processor systems, are described in Section 1.1, while Section 1.2 covers trends in parallel computing, i.e. multi-processor systems. In both cases, one of the most notable trends is the significant and growing gap between the theoretical performance limit of a computer system and the performance achieved in practice. Sections 1.1 and 1.2 describe factors that limit the performance of current single- and multi-processor systems, respectively. It will be shown that one of the most important factors in both cases is the latency of local and remote memory accesses. Since the inherent latency associated with local and remote memory accesses is rather large, latency reduction is only possible in a limited way. Therefore four common ways to tolerate latency instead of decreasing it are introduced in Section 1.3. Multithreading is the most general of these techniques in the sense that it makes the least assumptions about the applications and is further investigated in Section 1.4, which includes a detailed survey of multithreading implementations. Since none of the current commercial processors includes multithreading support in hardware, multithreading has to be implemented in software on these processors. Due to their coarse-grain context switches, current approaches at software multithreading are not suitable to hide the latency of local or remote memory accesses. Therefore a novel multithreading system is designed that enables fine-grain context-switches and is this able to tolerate the latency of these accesses. Section 1.5 summarizes the current chapter and provides an overview about the remaining chapters.

1.1 Trends in Sequential Computing

In the past 20 years, sequential computing has been dominated by microprocessors, i.e. processors implemented using very large scale or even higher levels of integration. The dramatic advances in semiconductor technology enabled the implementation of ever more complex microprocessors [HJ91]. This trend is illustrated in Figure 1.1, which depicts the number of transistors used

Fig. 1.1. Number of Transistors



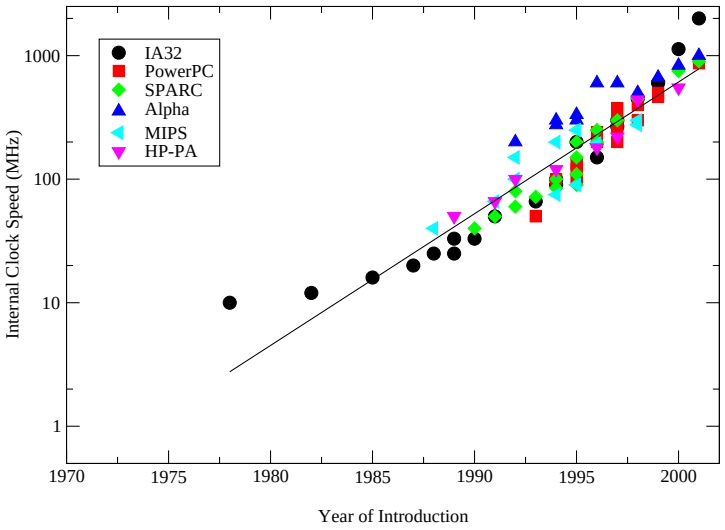
in implementations of common processor architectures. The figure is based on data from the CPU Info Center [Bur01]. Exponential regression testing on this data reveals that the number of transistors almost doubles each year, other published results range from 60 % to 80 % [HP96].

Apart from the higher levels of integration, the advances in semiconductor technology have also enabled a significant increase in the clock frequency of microprocessors. This trend is illustrated in Figure 1.2, which depicts the internal clock frequency for implementations of several common processor architectures. The figure is based on data from the CPU Info Center [Bur01]. Exponential regression testing on this data reveals that the clock frequency increases by 76 % each year, a trend that will likely continue in the foreseeable future.

The effect of the advances in semiconductor technology on both complexity and speed of microprocessors is illustrated in the above figures. However, the effect on the performance of microprocessors, as measured by the execution time of applications, is less pronounced. The performance of microprocessors is usually compared by measuring the execution time of certain benchmarks. The purpose of benchmarks is to predict the performance of a microprocessor in certain application areas. In other words, the performance of a microprocessor as measured by the execution time of the benchmarks should reflect the performance of the microprocessor running applications from these areas.

Benchmarks are usually small- to medium-sized applications that enable comparisons with other microprocessors. A detailed introduction to several popular benchmark suites can be found in Chapter 4. One of the most pop-

Fig. 1.2. Clock Frequency



ular benchmarks is the LINPACK benchmark, which solves a dense system of linear equations using LU decomposition [Don90]. The performance under the LINPACK benchmark as well as the theoretical possible performance for implementations of several common processor architectures is depicted in Figure 1.3. Although the LINPACK benchmark reflects only a restricted area of applications, it is well suited for long-term comparisons of performance, as LINPACK benchmark results are available for almost every known microprocessor [DS01]. Note that nowadays the SPEC benchmarks are used to compare the performance of microprocessors as these benchmarks provide a broader view of microprocessor performance.

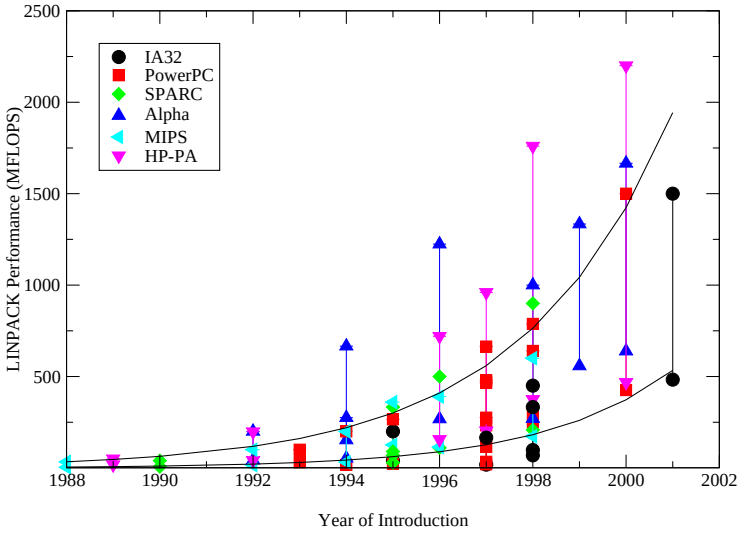
The LINPACK benchmark results as well as the theoretical performance limits are taken from [DS01]. Based on the data depicted in Figure 1.3, two different trends can be identified: On the one hand, the theoretical performance limit increases by 228 % each year on average. On the other hand, the performance achieved in practice, as measured by the LINPACK benchmark, increases by 204 % each year on average. The reasons behind this significant and growing gap are explained in the following paragraphs.

The maximum possible performance of a microprocessor is the internal clock frequency f_{clk} times the number of integer and floating-point instructions that can be issued in each cycle, i.e.

$$p\text{max} = f_{\text{clk}} \cdot (n_{\text{int}} + n_{\text{fp}})$$

measured in millions of instructions per second (MIPS). The above equation can be restricted to floating-point performance by considering only the number of floating-point execution units, i.e.

Fig. 1.3. LINPACK Performance ($n = 100$)



$$p\text{max} = f_{\text{clk}} \cdot n_{\text{fp}}$$

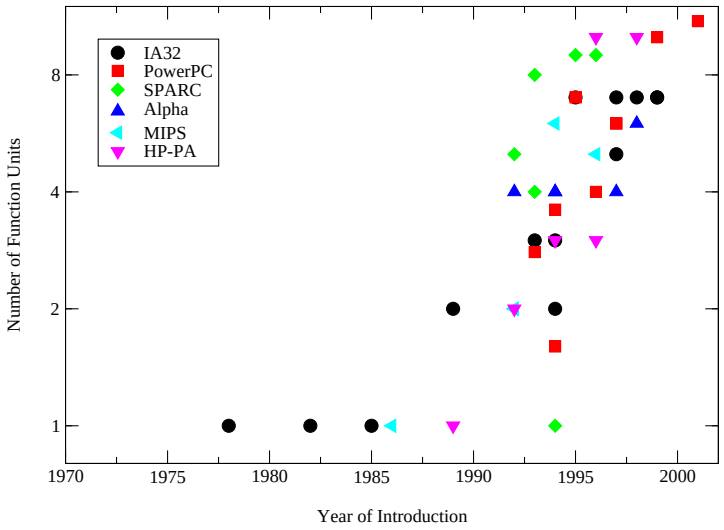
measured in million floating-point operations per second (MFLOPS). As already illustrated in Figure 1.2, the advances in semiconductor technology impact the clock speed of microprocessors. The increasing number of available transistors depicted in Figure 1.1 impacts the number of execution units, as Figure 1.4 shows.

Note that the above equations represent the theoretical performance limit under the assumption that all issue slots can be utilized in every clock cycle and all execution units are fully pipelined, i.e. can accept a new instruction in each cycle. In general, this assumption is not realistic, thereby rendering the above equations useless for predicting the performance of microprocessors on applications. One should think of these equations as representing the performance that a microprocessor is guaranteed not to exceed. The following sections describe some of the factors that limit the performance of current microprocessors.

1.1.1 Dependencies & Hazards

As stated above, the assumption that all execution units can be utilized, i.e. by executing $n_{\text{int}} + n_{\text{fp}}$ instructions in parallel in each clock cycle, is not realistic. The reason is the limited amount of available instruction-level parallelism in an application as well as the difficulty to extract this parallelism [Wal92][SJH89]. The instruction-level parallelism of a program is limited by instruction dependencies, which can be grouped into data, name, and control dependencies:

Fig. 1.4. Number of Function Units



- An instruction b is data-dependent on another instruction a, if either instruction a produces a result that is used by instruction b, or there exists an instruction c such that b is data-dependent on c and c is data-dependent on a. Note that the data-flow between two data-dependent instructions can occur either via registers or via memory locations. The former is easy to detect, but the latter requires alias analysis due to the large number of ways in which two instructions can access the same memory location.
- Two instructions a and b, where a is executed before b in sequential program order, are said to be anti-dependent if instruction b writes a register or memory location that instruction a reads. Sequential program order is the order in which instructions would be executed on a processor with a single function unit and in-order execution. Anti-dependency is one of the two forms of name dependency, the other form is output dependency: Two instructions a and b, where a is executed before b, are said to be output dependent, if both instructions write the same register or memory location. The difference between data and name dependencies is the absence of data flow between the two instructions.
- An instruction a is said to be control-dependent on a branch instruction b, if instruction b is on at least one path from the program entry point to instruction a in the static control flow graph of the program. Control dependencies restrict the reordering of instructions in two ways: Instructions that are control-dependent on a given branch instruction can usually not be moved before that branch, while instructions that are not control-dependent on a given branch instruction can usually not be moved behind that branch.

Two instructions that are independent on each other, i.e. are neither data-, name-, nor control-dependent can be executed in parallel. However, the reverse is not true: The existence of a dependency between these two instructions does not necessarily imply that executing those two instructions in parallel creates a hazard on a given processor. Dependencies are a property of the processor architecture, but the set of dependencies that lead to hazards are a property of the processor's implementation. There are three different sets of hazards, i.e. structural, data and control hazards.

Given a set of instructions that would execute in parallel, a structural hazard arises whenever one or more of the instructions cannot issue due to resource conflicts, e.g. lack of available execution units. Therefore structural hazards can arise even in the absence of dependencies.

Data hazards arise whenever a data or name dependency exists between instructions and the instructions are issued too close to one another. Given two instructions a and b, where a is executed before b in sequential program order, there are three different cases:

- A read-after-write hazard arises if instruction b reads a register or memory location before it is written by instruction a. If this kind of hazard is not resolved, instruction b will use the old value of the register or memory location.
- A write-after-write hazard arises if instruction b writes a register or memory location before it is written by instruction a. If this kind of hazard is not resolved, the register or memory location will hold the value written by instruction a instead of b.
- A write-after-read hazard arises if instruction b writes to a register or memory location before it is read by instruction a. If this kind of hazard is not resolved, instruction a will incorrectly use the new contents of the register or memory location.

A read-after-read situation, i.e. if instruction b reads a register or memory location before it is used by instruction a, is no hazard since the contents of the register or memory location are not modified by either of the instructions.

Control hazards are caused by control dependencies: Given two instructions a and b, where a is executed before b in sequential program order and b is control dependent on a, a control hazard arises if b is executed before the outcome of the branch a is known. Note that control dependencies are quite frequent due to the large number of branches in programs, i.e. on average, one out of six instructions is a branch [Wal92][SJH89].

Each of the hazards mentioned above will cause a pipeline stall upon its detection at runtime, i.e. the execution of one or more instructions must be delayed in order to resolve the hazard and ensure proper program semantics. Pipeline stalls can last for one or more clock cycles and affect one or more execution units such that these function units can perform no useful work for the duration of the stall. As observed in Figure 1.3, pipeline stalls can have a significant impact on the performance of a microprocessor. Therefore the

number and duration of pipeline stalls has to be decreased in order to close the gap between theoretical and practical microprocessor performance.

1.1.2 Dependency Removal

As pipeline stalls are caused by hazards which occur only in the advent of dependencies, with the exception of structural hazards, there are three ways to handle this problem: First of all the program can be changed such that the number of dependencies is reduced. By removing at least some of the dependencies, all hazards and pipeline stalls that might have been caused by these dependencies are effectively removed. Second, the internal architecture of the microprocessor, i.e. the pipeline, can be changed such that the number of hazards is reduced even in the presence of dependencies. Last, the pipeline can be changed such that the duration of pipeline stalls is reduced. The following paragraphs will describe each of these options in more detail.

The removal of dependencies is mostly performed by the compiler, although hardware support is required in some cases. The compiler has to analyze the existing dependencies before removing any of them. This dependency analysis is complicated by the presence of arrays and pointers in modern languages.

Pointers increase the complexity and introduce uncertainty in the analysis of dependencies due to aliasing, thereby causing conservative results. If the microprocessor supports dynamic memory disambiguation, i.e. resolving of conflicts due to aliasing, the compiler can perform more aggressive optimizations by ignoring some of the dependencies that arise from memory operations.

Arrays increase the complexity of dependency analysis by making it hard to determine the dependencies inside loops. In order to determine whether two loop iterations are independent, constrained Diophantine equations based on the array indices have to be solved. Unfortunately, this problem is equivalent to integer programming, which is known to be NP-complete [MHL91]. However, most of the equations that occur in practice are quite simple, thereby enabling the use of simple tests that determine whether a dependency exists. Once the dependencies have been determined, the compiler uses techniques like register renaming, loop unrolling, software pipelining, trace scheduling or speculation to reduce the number of dependencies.

Register renaming [Sim00] eliminates name dependencies by changing the conflicting registers. Note that name dependencies occur via registers as well as memory locations, but renaming is done more easily for register operands. Register renaming can be performed statically by the compiler or dynamically by the microprocessor. However, the ability to remove name dependencies is restricted by the limited number of available registers, hence modern microprocessors provide more registers than the corresponding architecture requires, thus increasing the benefits of register renaming.

Loop unrolling reduces the number of control dependencies by replicating the loop body several times and modifying the loop header accordingly. In addition, loop unrolling increases the potential for instruction scheduling as the instructions from different unrolled iterations of the loop must be independent, otherwise loop unrolling would not be not possible.

Software pipelining [RGSL96] is related to loop unrolling and is sometimes called symbolic loop unrolling. Software pipelining reduces the number of data dependencies inside loop iterations by interleaving instructions from different iterations of the loop. Loop unrolling can be used to increase the size of the loop body, thereby increasing the effectiveness of software pipelining. However, the management of registers in software pipelined loops can be quite complex.

Trace scheduling [Fis81] reduces the number of control dependencies by selecting a set of basic blocks that are likely to be executed in sequential order and creating a larger basic block by omitting the branches between the original basic blocks. Static branch prediction is used to determine the likelihood that a set of basic blocks is executed in sequential order. In contrast to loop unrolling, trace scheduling is not restricted to loop branches. However, trace scheduling complicates the code due to the need for additional bookkeeping code that handles mispredictions.

Speculation [EGK+94] reduces the number of control dependencies by predicting the outcome of branches and by moving instructions across the controlling branch. However, the speculative code must not destroy program semantics even in the presence of mispredictions, hence the compiler has to make conservative decisions about which instructions to move. The efficiency of speculation is significantly increased in the presence of hardware support, i.e. the microprocessor ensures that speculated instructions do not commit until they are no longer speculative.

Finally, conditional or predicated instructions can be used to transform control dependencies into data dependencies. Conditional instructions evaluate a certain condition and behave like a null operation if this condition is not true. The most common form of conditional instructions is the conditional move between registers, which is supported by all modern architectures. Conditional instructions can be used to eliminate control dependencies by transforming them into data dependencies, and to protect instructions that are moved across controlling branches. However, conditional instructions are often slower than their unconditional counterparts and still consume execution time, even if the tested condition does not hold.

Dependencies that cannot be eliminated by the techniques presented above do not necessarily cause hazards. Apart from instruction scheduling, most of the techniques used to avoid hazards in the presence of dependencies involve changes to the internal architecture of the microprocessor: dynamic scheduling, branch prediction, and caches.

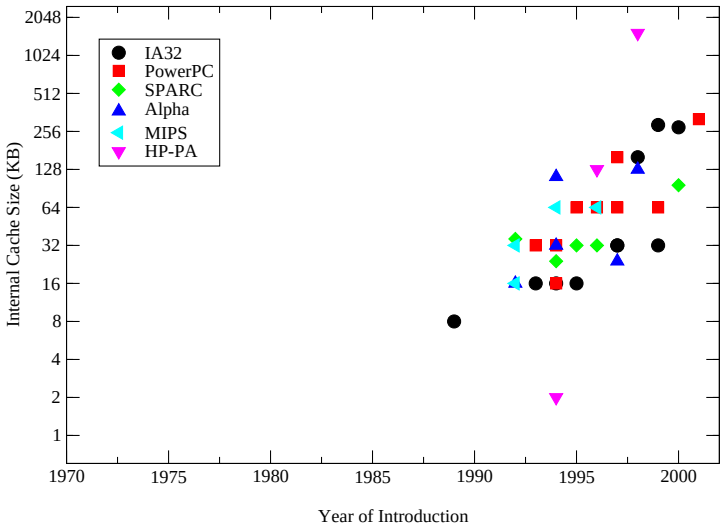
The compiler uses instruction scheduling to rearrange instructions in a basic block such that pipeline stalls are minimized. Instruction scheduling uses the available parallelism between independent instructions to hide the latency of instructions that might cause a pipeline stall. However, it becomes more and more difficult to find the required number of independent instructions inside a basic block as the latency to be hidden grows larger, hence more aggressive scheduling has to be used in these cases.

Dynamic scheduling rearranges the execution of instructions at run-time in order to prevent pipeline stalls. The instructions are issued in sequential order as long as no structural hazards exist, but the individual instructions are executed as soon as their operands become available, i.e. out-of-order. Even if some instructions are stalled due to data dependencies, subsequent instructions can be executed as long as no structural hazards arise. Note that out-of-order execution implies out-of-order completion and is used by almost all recent implementations of modern processor architectures. Out-of-order execution is especially effective when combined with register renaming. However, out-of-order execution requires a lot of resources and complicates the pipeline control logic, e.g. to maintain precise interrupts.

Branch prediction is used by modern microprocessors to reduce the impact of control dependencies: Additional hardware is used to predict the outcome and/or target of branches from previous executions of the branch. A survey of popular branch prediction strategies is included in Section 2.3. Although the accuracy of current branch prediction schemes is quite good, modern microprocessors have to predict several consecutive branches in order to avoid pipeline stalls, as branches occur frequently in typical instruction streams. The combined accuracy for a sequence of branches is given by the product of the accuracies of the predictions for the individual branches, i.e. decreases significantly for larger sequences.

Caches are used to decrease the latency associated with memory operations. Recall that the latency of instructions is one of the problems that keep instruction scheduling from avoiding pipeline stalls. Caches are a part of the memory hierarchy, a chain of memories of increasing size and latency with growing distance from the microprocessor. Section 2.3 describes caches and their properties in detail. Each memory in the hierarchy usually contains a subset of the contents of the memory on the next level, although there are exceptions to this rule, e.g. exclusive caches. Frequently used memory locations should reside in the upper levels of the hierarchy, i.e. next to the microprocessor. Caches exploit the principles of locality observed in programs: It is likely that accessed memory locations will be accessed again in the near future, i.e. temporal locality. It is likely that locations in the vicinity of accessed memory locations will be accessed in the near future, i.e. spatial locality. However, the effectiveness of caches depends on the particular program, e.g. programs with irregular access patterns will seldom benefit from caches.

Fig. 1.5. Internal Cache Size



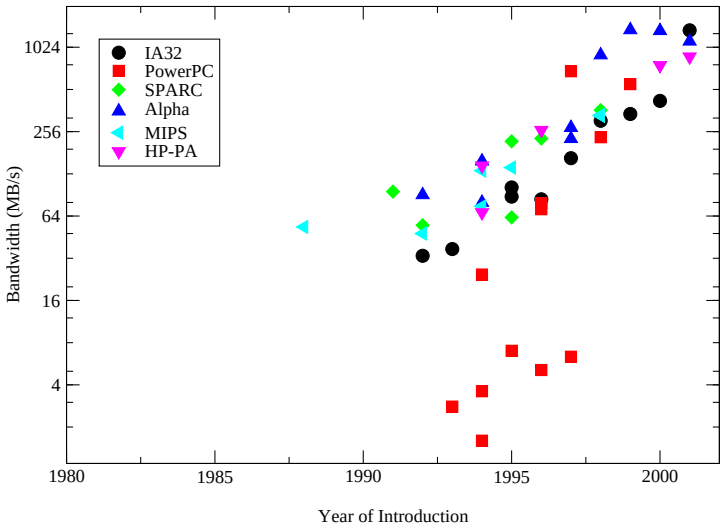
If the techniques presented above fail and a pipeline stall is unavoidable, the pipeline should be designed such that the length of the pipeline stall is minimized. This is accomplished by result forwarding, i.e. providing bypasses between different stages of the pipeline. Result forwarding reduces the length of pipeline stalls, as the results are passed directly to dependent instructions in the pipeline instead of passing the results via the register file. Another way to avoid or minimize the length of pipeline stalls is to provide enough resources, such that structural hazards are avoided. These resources include execution units, register file read and write ports, cache ports as well as the units that control the pipeline. However, these resources increase the complexity of the microprocessor, thereby potentially reducing the maximum clock frequency.

1.1.3 Caches & Main Memory

As already mentioned, caches are used to reduce the potential for pipeline stalls arising out of data dependencies involving main memory. As Figure 1.5 illustrates, caches use a significant portion of the transistor budget in current microprocessors, partially because those transistors could not be usefully exploited otherwise.

Figure 1.5 depicts the size of the internal caches for implementations of several processor architectures. Although the size of the transistor budget is steadily increasing, the size of the internal caches increases even faster, i.e. the percentage of the transistor budget attributed to the internal caches is increasing. Since caches seem to be an important part of modern micropro-

Fig. 1.6. Memory Bandwidth according to the STREAM Benchmark



cessors, the following section provides a detailed look at main memory, whose latency has to be hidden by caches.

Main memory is usually made from DRAM (dynamic random access memory), while caches are usually made from SRAM (static random access memory). DRAM uses a single transistor to store each bit, SRAMs use between four and six transistors for each bit. DRAMs are usually slower than SRAMs and have to be refreshed periodically to maintain the contents of the memory array. On the other hand, DRAMs have a larger capacity, consume less power and are cheaper than SRAMs. Independent of the type of the memory, the performance of a memory system can be expressed in two quantities: bandwidth and latency.

Bandwidth. Bandwidth refers to the maximum rate at which the memory can deliver data, usually given in megabytes or gigabytes per second. The STREAM benchmark [McC95] is often used to measure the bandwidth of main memory. Figure 1.6 depicts the bandwidth achieved in practice for computer systems based on the implementations of several processor architectures as measured by the STREAM benchmark. Note that the bandwidth and latency of the main memory system depends on the microprocessor as well as the surrounding system. However, there are several ways to increase the bandwidth of main memory: increasing the width of the memory bank, interleaving multiple banks of memory, and using multiple independent banks of memory.

Given a main memory system using a data bus width of n bytes, i.e. the bus can transfer n bytes per clock cycle, yielding a bandwidth b of

$$b = n * f_{\text{clk}}$$

bytes per second. According to this equation, multiplying the width of the data bus by k increases the bandwidth of the memory system by a factor k as long as the width of the individual memory banks is increased by the same factor. However, the associated cost restricts the practical width of the memory bus: Personal computers usually use an 8 byte wide memory bus, while workstations use a memory bus that is up to 16 bytes wide.

Interleaving uses multiple banks of memory that can be accessed in parallel. Each bank has the same width as the memory bus, addresses are usually interleaved at the word level, i.e. the ith bank contains all memory locations with addresses of the form

$$n \cdot k + i \qquad \forall n \in N, 0 \leq i < k$$

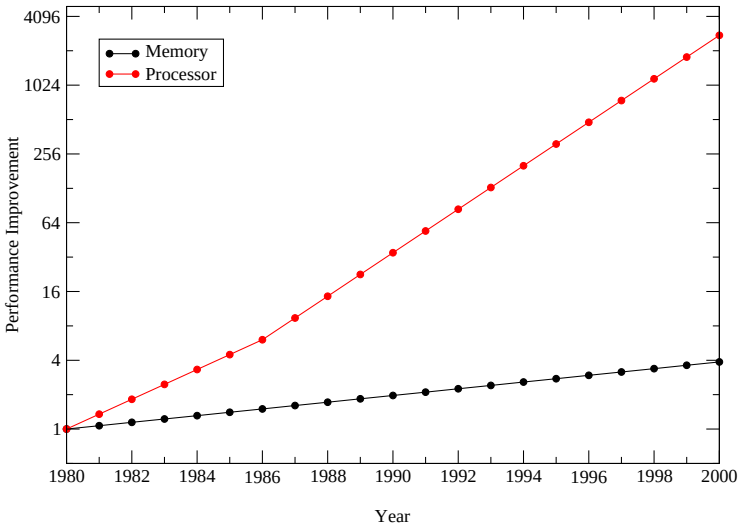
Such a mapping is useful for sequential accesses, e.g. cache fills, since all banks are accessed in parallel, such that k consecutive words are available simultaneously. However, the consecutive words have to be transferred sequentially, since the width of the memory bus equals the width of each memory bank. A drawback of interleaving are the associated costs: All banks have to be populated, hence a k-way interleaved memory system uses at least k memory modules. A traditional memory system uses fewer modules of larger size for the same amount of memory, which is usually more cost-effective.

Another form of interleaving exploits the internal structure of DRAMs to reduce the latency and thereby increase the bandwidth: Due to pin-count restrictions, DRAMs use a multiplexed address bus to access the internal memory array. Each memory location is accessed by subsequently specifying the corresponding row and column addresses. Since the row address is transferred first, subsequent accesses to locations in the same row only need to specify another column address, thereby reducing the latency of the access. This type of access is called page-mode and is supported by all DRAMs since the 1 Mbit generation.

Independent memory banks are a generalization of memory interleaving: Like before, the memory system consists of multiple memory banks. The difference between interleaving and independent memory banks is that the latter uses separate address and data busses for each memory bank, thereby allowing independent accesses to each bank. Note that memory interleaving uses a shared address and data bus, i.e. all memory banks are accessed in parallel at the same address. The mapping of addresses to the independent memory banks is usually the same as in the interleaved approach.

As long as memory accesses go to different memory banks, such a memory system is very effective. Given an even number of memory banks, sequential accesses as well as accesses that are separated by an odd number of memory locations will access different banks. However, accesses that are separated by an even number of memory locations will access the same bank if the difference is a multiple of the number of banks. This problem can be solved by providing a large number of memory banks, thereby reducing the chances that

Fig. 1.7. Performance Improvements for Processors and Memory



two consecutive accesses hit the same bank. The drawback of this approach is the associated cost, hence this approach is only used in some supercomputers. Another solution is to change the access patterns of the program, e.g. by having the compiler expand the size of arrays.

Latency. Latency refers to the length of time that the memory needs to deliver the contents of a single memory location and is usually given in nanoseconds. Despite advances in semiconductor technology, memory access times have not kept up with the improvements in microprocessor clock speed, as Figure 1.7 illustrates. Figure 1.7 is taken from [HP96] and depicts the improvements of microprocessor clock frequency and main memory access times, using 1980's performance as a baseline. Note that the number of clock cycles that elapse during a memory access, has increased as well, making it difficult at best to avoid pipeline stalls due to data dependencies involving main memory. The use of a memory hierarchy with one or more caches is a consequence of this performance gap, since caches use replication of frequently used memory locations in order to avoid accesses to main memory.

Given a memory hierarchy with caches, the average memory access time is determined by several parameters: The hit time is the amount of time required to retrieve the contents of a memory location that is found in the cache, while the miss penalty is the amount of time required to retrieve the contents of a memory location that is not found in the cache. The percentage of accesses to the caches that result in a cache miss is the miss rate. According to [HP96], the average memory access time a is given by :

$$a = \text{Hit time} + \text{Miss Rate} \cdot \text{Miss Penalty}$$

Based on the above equation, three different ways to decrease the average memory access time can be identified: decreasing the access time in case of cache hits, decreasing the ratio of cache misses, and decreasing the penalty associated with cache misses. Several factors affect the miss rate, e.g. cache line size, cache associativity, hardware and software prefetching, and compiler optimizations. Other factors affect the miss penalty as well as the hit time. A detailed discussion of these factors is outside the scope of this chapter, Section 2.3 includes an introduction to caches. Detailed informations about the design of caches can be found in Handy's book [Han93].

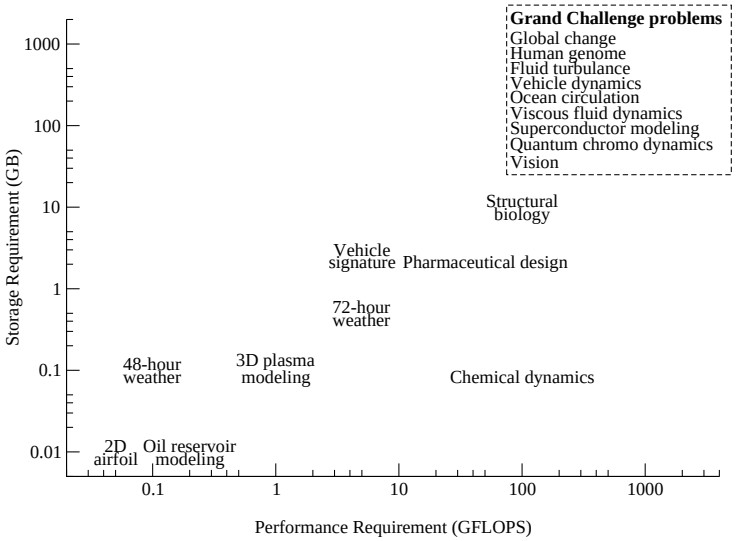
Caches are an integral part of modern microprocessors, since the processor-memory performance gap is so large that accesses to main memory frequently cause pipeline stalls. The importance of caches is reflected by the fact that a huge portion of the transistor budget in current microprocessors is dedicated to caches. However, the effectiveness of caches largely depends on the software: Some programs show regular access patterns that are cache-friendly, other programs show irregular access patterns. As the processor-memory performance gap will continue to increase for the foreseeable future, the latency of accesses to main memory is one of the major bottlenecks that limit the performance of current microprocessors.

1.2 Trends in Parallel Computing

The performance of microprocessors is steadily increasing, in spite of the limiting factors identified in the previous section. However, the computational performance required by some important applications exceeds the capabilities of even the fastest microprocessors. Some of these so-called Grand Challenge Problems were identified by the Committee on Physics, Mathematics. and Engineering Sciences of the federal Office of Science and Technology (OSTP). Figure 1.8 is taken from [CS99] and summarizes the findings of this committee by characterizing several important applications from science and engineering by their computational demands and storage requirements. Note that new challenges will be added as computer performance increases and new applications become feasible.

Parallel computers use several processors to achieve an aggregate performance that is equal to the number of processors times the performance of a single processor, at least in principle. The Grand Challenge Problems mentioned above require machines with a very large number of processors, so-called massively parallel processors (MPP). These MPPs have traditionally been custom-built machines based on commercial microprocessors that were tightly integrated such as the Cray T3E [Oed96]. Clusters are a new class of massively parallel processor that use commercial off-the-shelf workstations or servers as computing nodes in connection with a custom-designed network. As the network interfaces are based on standard busses, the microprocessors are less tightly integrated than in traditional MPPs. However,

Fig. 1.8. Grand Challenge Problems



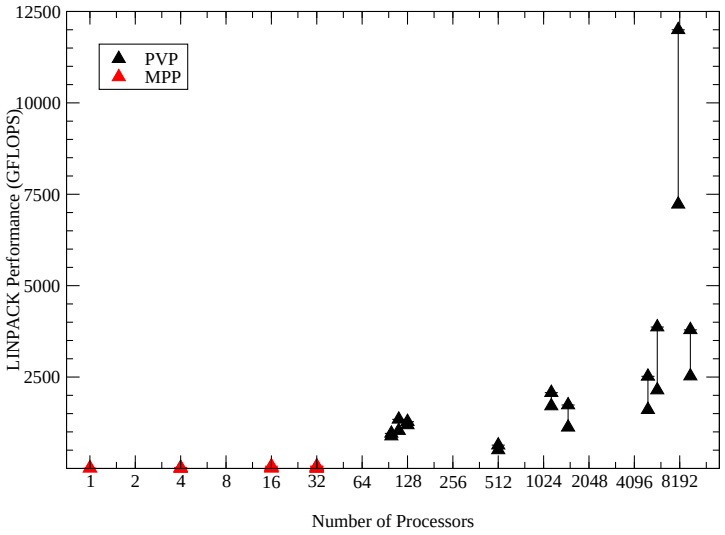
clusters provide a good price/performance solution as the are mostly based on commercial off-the-shelf technology. The Compaq AlphaServer SC [Cor00] is an example for such a cluster.

Although current MPPs can support several thousand processors, most installed MPPs use several hundred processors. These large number of processors yield impressive maximum performance figures. However, the performance achieved in practice often deviates from the theoretical maximum performance. The situation in parallel computing is therefore similar to the situation of sequential computing that was described in Section 1.1, although the effects are even more pronounced. Figure 1.9 depicts the performance achieved in practice as measured by the LINPACK benchmarks as well as the theoretical maximum performance for several parallel computer systems. In contrast to Figure 1.3, Figure 1.9 denotes the performance in GFLOPS instead of MFLOPS.

As the processing nodes of massively parallel processors are built around commercial off-the-shelf microprocessors, part of the performance gap can be attributed to the performance limiting factors described in Section 1.1. However, some performance limiting factors are unique to parallel computing. The three most important factors in this area are parallelization, load balancing, and communication.

If a program is not perfectly parallelizable, i.e. is always able to utilize all available processors, the performance is limited by Amdahl's law: Assume that a given program has an execution time t_{seq} on a single-processor system and that a fraction α of the program can be parallelized using all p processors on a multi-processor system. According to Amdahl's law, the Speedup s, i.e. the factor by which performance increases from using p processors is given

Fig. 1.9. Parallel LINPACK Performance



by

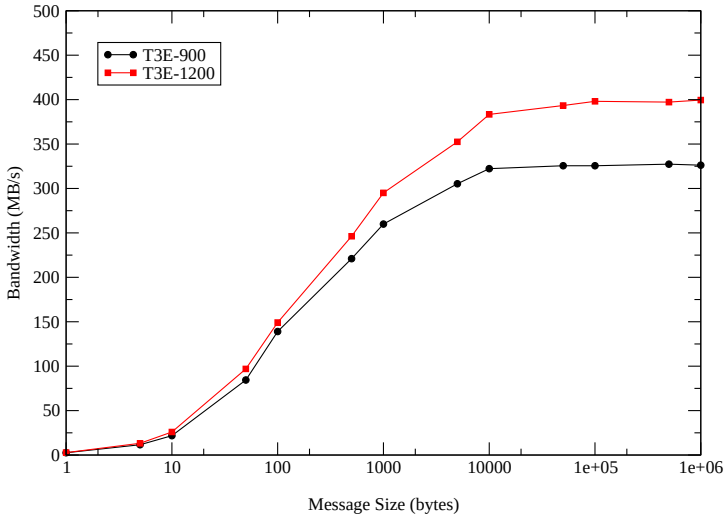
$$s = \frac{1}{(1-\alpha) + \frac{\alpha}{p}}$$

Note that the speedup is effectively bounded by $1/(1-\alpha)$, i.e. independent of the speedup achieved in the parallel section of the program. For example, even if 90 % of the program executes in parallel, the maximum speedup will be less than ten instead of the desired speedup of p. Therefore programs have to be completely parallel in order to yield competitive speedups on large-scale MPPs with hundreds or thousand of processors. This problem applies to parallel computing in general and is independent of the architecture of the parallel computer.

The second problem, improper load balancing, arises even if the program is fully parallelizable: The desired speedup of p can only be achieved if the workload is evenly distributed among the p processors. Therefore load balancing is an important property of parallel programs. Load balancing is complicated in two ways. On one hand, it might be difficult to divide the workload into individual tasks, such that at least one task is assigned to every processor. On the other hand, it might be difficult to determine the complexity of a given task in advance. Nevertheless, several load balancing techniques have been developed that work reasonably well in practice. Like before, this problem applies to parallel computing in general and is independent of the architecture of the parallel computer.

The third problem is the amount and type of communication between processors. Communication arises whenever a processor needs to access data that is stored in the memory of another processor. The obvious way to avoid

Fig. 1.10. Remote Memory Bandwidth in the Cray T3E

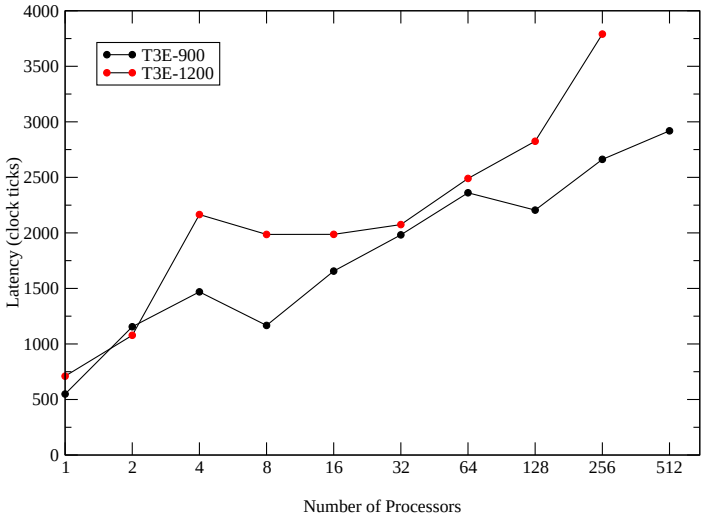


communication is to distribute the data across the processors, such that each processor owns the data that it frequently accesses. Even if such a data distribution can be found, parallel algorithms possess an inherent amount of communication. The impact of communication on performance depends on the architecture of the parallel machine, especially the network that connects the individual processors. The performance of the network can be defined in the same terms that were used in Section 1.1 to define the performance of the memory system, i.e. bandwidth and latency. In fact, the memory of another processor can be seen as an additional level to the existing local memory hierarchy. As Figures 1.10 and 1.11 show, the bandwidth and latency of communication links in current massively parallel processors are considerably worse than the bandwidth and latency of the local memory system, i.e. providing an even larger potential for pipeline stalls due to data dependencies.

Figure 1.10 depicts the bandwidth of the communication links as a function of the block size for two different models of the Cray T3E, i.e. the T3E-900 and the T3E-1200. Note that the maximum bandwidth of the communication links is 327 MB/s for the T3E-900 and 399 MB/s for the T3E-1200. Figure 1.10 suggests that parallel programs should use large communication blocks whenever possible to benefit from the higher bandwidth.

Figure 1.11 depicts the latency in terms of processor clock cycles of reading a memory location on a remote processor as a function of the machine size for two different models of the Cray T3E, i.e. the T3E-900 and the T3E-1200. Note that the corresponding measurements were performed in batch mode and are affected by the allocation strategy of the NQS batch queuing system: The allocated partitions for a given batch job can be non-contiguous, i.e.

Fig. 1.11. Remote Memory Latency in the Cray T3E



the distance between the individual processing elements is larger than the minimum distance for the given number of processors. Unfortunately, it is not possible to influence the allocation strategy short of running the whole machine in dedicated mode.

The processor used in the T3E issues up to four instructions per clock cycle, hence it is unlikely that a program contains enough instruction level parallelism to avoid pipeline stalls in the presence of latencies as shown in Figure 1.11. Unfortunately, there is an inherent latency for the communication links due to the physical size of the machine: the larger the number of processors, the larger the physical size of the machine and therefore the inherent latency, as electric signals propagate with limited speed.

Section 1.1 identified the latency of main memory as one of the most important bottlenecks that limit the performance of current microprocessors. In the case of parallel computing, the bottleneck due to the latency of the communication network is even more pronounced. As opposed to bandwidth, latency cannot be decreased below the inherent latency by using more resources. Therefore Section 1.3 describes several techniques for reducing and tolerating long-latency events such as main memory accesses or communication.

1.3 Latency Tolerance

The previous two sections have identified the latency of the memory hierarchy as one of the most important bottlenecks that limit the performance of single-

and multi-processor systems. There are three ways to reduce the latency of the memory hierarchy: First of all, the access time at all levels of the memory hierarchy can be reduced by careful design. Although the inherent latency can not be reduced in any way, one can try to design a memory hierarchy that does not exceed the inherent latency too much. The second approach tries to avoid long-latency accesses by using replication, i.e. providing copies of frequently accessed data at lower levels of the memory hierarchy. As long as the application possesses temporal and/or spatial locality, this approach can be very effective. However, not all applications have this property, e.g. irregular applications. Last but not least, the application itself can be restructured, such as to reduce the frequency of long-latency accesses and/or improve the access patterns to better suit automatic replication. A survey of latency reducing techniques can be found in [GHG+91]. Although these techniques can be very effective, the remaining latency is still large enough to impact the performance of single- and multi-processor systems.

In addition to of reducing the latency as discussed in the previous paragraph, one can try to tolerate the remaining latency, thereby reducing the impact on performance. There are four different ways to tolerate long-latency events: block data transfer, prefetching, asynchronous accesses and multithreading. A survey of latency tolerance techniques can be found in [MCL98][GHG+91].

Block data transfer [WSH94] uses a smaller number of large messages instead of a large number of small messages by combining communication requests whenever possible. This approach benefits from the higher bandwidth that can be achieved for large messages as well as the reduced per-message overhead. Although the latency of the first message word is not reduced at all, the subsequent message words arrive in short increments due to the high bandwidth available for large messages. Block data transfer is effective as long as the application is well-suited for this approach, i.e. has enough potential for combining communication requests. However, even if block data transfer is used, the processors will stall until the first message word is received. The remaining three approaches address this problem as well and are complementary to block data transfer.

Prefetching [LM96][CB94] uses a split-transaction protocol for communication requests: The communication is initiated before the data is actually used, at which point the communication is completed. If the distance between initiation and completion is large enough, the message data is at least partially available before it is used, i.e. the latency of the communication is at least partially hidden by the instructions between initiation and completion. Note that the initiation must not stall the processor, otherwise no latency hiding is possible. The major drawback of prefetching is the fact that the target of the communication request must be known a sufficient amount of time before the data is actually used. The effectiveness of prefetching therefore depends largely on the application. Prefetching can either be implemented in

software, i.e. the compiler inserts corresponding prefetching instructions, or in hardware, i.e. the processor prefetches data automatically.

Asynchronous communication [CS99] uses communication primitives that do not stall the processor, i.e. the processor can perform independent work while the communication is pending. Asynchronous communication is similar to prefetching in that it uses independent instructions to hide the latency of communication. However, prefetching finds these independent instructions before the point where the requested data is actually used, while asynchronous communication finds independent instructions after this point. The advantage of asynchronous communication over prefetching is that it is no longer necessary to determine the target of the communication request in advance. But as communications are usually performed just before the data is needed, it may be difficult to find enough independent instructions. Again, the effectiveness of asynchronous communication depends largely on the application.

Multithreading [KCA91][BR92] is similar to asynchronous communication in that it uses communication primitives that do not stall the processor until the data is actually needed. In contrast to asynchronous communication, the independent instructions that are used to hide the latency of the communication request are not taken from the same thread of computation, but from another thread on the same processor. Therefore the processor has to execute several threads in parallel, and to switch between these threads in order to hide long-latency events. Multithreading is less susceptible to application behavior as the independent instructions are taken from other threads, i.e. is not restricted by the limited amount of instruction level parallelism that is available in a single thread.

All four techniques presented above are effective in hiding long-latency events. However, these techniques have significant costs: As the first technique combines communication requests and the other three techniques use independent instructions to hide the latency, the application has to possess additional parallelism. In addition, all techniques increase the bandwidth requirements of the applications, as the same amount of communication is performed in less time. At least the last three techniques complicate the hardware, e.g. by supporting multiple outstanding requests or multiple threads. It is unlikely that enough independent instructions can be found in a single thread to tolerate the latencies encountered in massively parallel processors. Multithreading is the only technique that addresses this problem and is therefore investigated further in Section 1.4. In addition, multithreading can be combined with block transfer and prefetching to yield even better performance.

1.4 Multithreading

A conventional single-threaded architecture executes a single thread of control, i.e. instructions from the corresponding program are executed in sequential order. The state of such a system consists of two parts: memory state and processor state. The memory state refers to the contents of main memory, i.e. program code, stack, data. The processor state consists of the activity specifier, i.e. stack pointer and program counter, as well as the register context, i.e. the contents of the architected registers. In the case of a single-threaded architecture, the processor state is restricted to the context of a single thread.

In contrast, a multithreaded architecture executes instructions from several threads of control, i.e. the instruction to be executed is chosen from several candidates, one from each thread. As a consequence, the processor state of such an architecture consists of several activity specifiers and register contexts, i.e. thread contexts, while the memory state contains several stacks and may contain different program code.

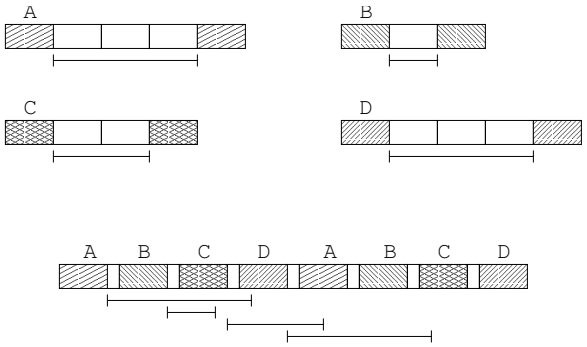
Multithreading can be used to tolerate long-latency events by switching to another thread whenever the current thread encounters such an event. With any chance, the results of the long-latency operation are available the next time that the thread is activated, thereby effectively hiding the latency by executing other threads in the meantime. Since context switching itself takes some time, the context switch overhead influences the amount of latency that can be hidden effectively. An example of this situation is illustrated in Figure 1.12.

In this example, there are four threads named A through D. The upper part of the figure shows how these threads would be executed on a single-threaded architecture. Note that useful work performed by the individual threads is marked in different shades of grey, while white represents idle or stall cycles. The lower part of the figure shows how these threads would be executed on a multithreaded architecture, if context is switched whenever a long-latency operation is encountered and threads are scheduled round-robin. Note that it is useless to initiate a context switch upon encountering any long-latency event: If the latency of the event is less than two times the context switch time.

Multithreading can be implemented in hardware using custom-designed processors or in software using commercial off-the-shelf processors. Section 1.4.1 surveys multithreading systems implemented in hardware, while Section 1.4.2 covers software implementations.

Multithreaded processors implement multithreading in hardware by extending conventional processors with support for multithreading, e.g. by adding multiple program counters and/or register contexts. Therefore multithreaded processors follow the control-flow computing model like their conventional counterparts. This restricted definition of multithreaded processors follows the definition in [vRU99] and is used to distinguish these architectures from other architectures that follow the data-flow computing model,

Fig. 1.12. Latency Tolerance via Multithreading



e.g. large grain data-flow or threaded data-flow. Note that the term multithreaded architecture is often used in both cases.

The number of threads that can be executed on a multithreaded processor is usually equal to the number of supported program counters and/or register contexts. However, the number of threads can be extended by using some form of software multithreading on top of the hardware implementation. The capabilities of the hardware threads is determined by the software that creates the individual threads, the hardware itself usually poses no restrictions on thread capabilities.

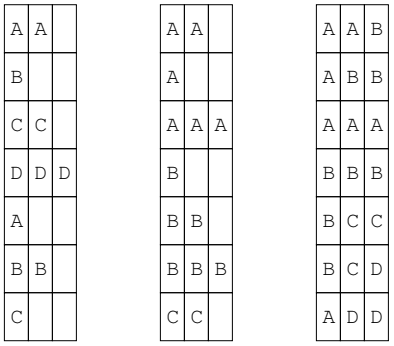
1.4.1 Hardware Multithreading

The most notable difference between multithreaded processors is the context switch strategy that can be used to group the individual processors into three classes: interleaved multithreading, block multithreading and simultaneous multithreading. The three different approaches are depicted in Figure 1.13 and are discussed in the following sections.

Interleaved Multithreading. Multithreaded processors based on cycle-by-cycle interleaving perform a context switch after each instruction fetch. As long as the number of threads exceeds the number of pipeline stages, there is at most one instruction from each thread in the pipeline, i.e. control and data dependencies are eliminated. Due to the absence of dependencies, the pipeline itself is simple, i.e no pipeline interlocks or forwarding paths are necessary. Another advantage of the cycle-by-cycle interleaving model is the absence of any context switch overhead, as context switches are well known in advance.

The major drawback of this model is the lack of single-thread performance: If only one thread is executed, the maximum performance of the processor is degraded by a factor that is equal to the number of pipeline

Fig. 1.13. Context Switch Strategies for Hardware Multithreading



stages. However, this problem can be addressed by using dependence lookahead or interleaving [LGH94]. The former approach issues sequences of instructions from the same thread as long as these instructions are neither data- or control-dependent. The instruction sequences are identified by the compiler, the instruction format uses an additional field to store the length of the sequence. The latter approach subsequently issues several instructions from the same thread as well, but extends the pipeline to support forwarding paths and pipeline interlocks instead. The following paragraphs describe several processors using interleaved multithreading.

The HEP (Heterogeneous Element Processor) [Smi78][Smi81a] is a shared memory multiprocessor that supports up to 16 processors, 128 memory modules, 4 I/O modules, 4 I/O cache modules, as well as an additional I/O processor. The individual elements are connected by a switch network made from three-port switches. Each processor supports up to 128 threads that share a single register file. Synchronization between threads is supported by full/empty bits for every memory location.

The Horizion [TS88][KS88] is a shared memory multiprocessor based on the earlier HEP design, that supports up to 256 processors and 512 memory modules. The individual elements are connected by a three-dimensional torus network made up from seven-port switches. Each processor supports up to 128 threads that share a single register file. Synchronization between threads is supported by full/empty bits for every memory location. In contrast to HEP, the Horizon was never implemented.

The Tera MTA (Multithreaded Architecture) [ACC$^+$90][AKK$^+$95] is a shared-memory multiprocessor based on the earlier HEP and Horizon designs, that supports up to 256 processors, 512 memory modules, 256 I/O cache modules, as well as 256 I/O processors. The individual elements are connected by a three-dimensional torus made from up to 4096 five-port switches. Each

processor supports up to 128 threads and is based on a three-issue long-instruction-word (VLIW) architecture with explicit dependence lookahead. The Tera MTA is a commercial product, but only one system has been sold.

The MASA (Multilisp Architecture for Symbolic Applications) [HF88] processor architecture is targeted at efficient execution of parallel LISP programs, e.g. MultiLisp. The processor is based on a load/store RISC architecture that was extended to support multiple threads and tagged data. Synchronization between threads is supported by full/empty bits for every memory location.

The MAP (Multi-ALU Processor) [KDM+98] used in the MIT M-Machine [FKD95] is based on a three-issue VLIW architecture. The Multi-ALU processor consists of three multithreaded clusters, each clusters supports up to four threads in hardware as well as four software threads per hardware threads, i.e. up to 16 threads in total. Processor coupling [KD92] is used for efficient synchronization between threads.

The MediaProcessor [Han96] is targeted at multimedia applications and supports up to five threads in hardware. The instruction set architecture is based on a load/store RISC architecture, with additional support for group instructions, e.g. shuffling, swizzling and permutation.

The SB-PRAM [ADK+92][BBF+97] implements the concurrent read, concurrent write (CRCW) parallel random access machine (PRAM) programming model and supports up to 256 processing elements, that are connected to the memory modules by a bidirectional butterfly network . Each processing element supports up to 32 threads in hardware, each threads supports another 32 software threads, i.e. 4096 threads in total.

Block Multithreading. Multithreaded processors based on the block interleaved model execute a single thread until a context switch is triggered, usually by an long-latency event. The context switch can be triggered either statically or dynamically.

In the static approach, context switches are associated with specific instructions, i.e. a context switch is performed whenever one of these instructions is executed. These instructions can trigger a context switch either implicitly or explicitly: In the former case, special context switch instructions are added to the instruction set and inserted into the instruction stream by the compiler. In the latter case, context switches are associated with certain groups of instructions, e.g. loads, stores, branches. The advantage of the static approaches is the low context switch overhead, as context switches are detected early in the pipeline, i.e. the instruction fetch stage. In the dynamic approach, context switches are triggered by certain events, e.g. cache misses, signals, or condition codes. As these events are detected rather late in the pipeline, the context switch overhead is increased by the need to flush all subsequent instructions in the pipeline.

Compared to the cycle-by-cycle interleaved model, the block interleaved model allows single thread performance comparable to conventional proces-

sors, at the expense of increased complexity due to the traditional pipeline as well as increased context switch overhead in the case of the dynamic approach. The following paragraphs describe several processors using block multithreading.

The Sparcle [Aga92] processor implements the APRIL [ALKK90] processor architecture used in the MIT Alewife [ABC+95] distributed shared memory machine. The Sparcle processor is based on the SPARC architecture and was extended to support up to four threads as well as fine-grained synchronization. Context switches are initiated if a remote cache miss or a synchronization event is encountered and has an overhead of 14 clock cycles.

The MSPARC [MD96] processor is similar to Sparcle: The processor is based on the SPARC architecture, supports up to four threads and performs a context switch upon cache misses. However, the context switch overhead was reduced to one cycle plus an additional four cycles for refilling the pipeline, i.e. a total of five cycles.

The Rhamma [GU96][GU97] processor uses a decoupled architecture [Smi82], i.e. separates execution and load/store units. The execution unit is based on the DLX [HP96] architecture and performs a context switch whenever a load, store, or branch instruction is executed, i.e. uses a static approach. The load/store unit performs a context switch whenever an instruction, that is not a load or store instruction, is encountered , i.e. uses a static approach. In addition, both units perform a context switch if operand values are unavailable or an explicit switch instruction is encountered and a certain condition is met, i.e use dynamic approaches as well. Both units access the same register contexts and are connected by first-in, first-out queues to pass threads. The Rhamma processor uses a context switch buffer to reduce the context switch overhead of one cycle in the case of a switch due to a load or store instruction. In all other cases, the context switch overhead is already zero.

The MARS-M (Modular, Asynchronous, Expandable, Multithreaded System) [DW92] is a shared-memory heterogeneous processor that consists of a control processor, a central processing unit, and a multi-ported memory. The control processor supports up to eight threads and contains eight execution units. The central processing unit uses a decoupled architecture, i.e. separate address and execution processors. The address processor supports four threads and contains twelve execution units, while the execution processor supports four threads and contains ten execution units. The MARS-M architecture inspired the MTD (Multithreaded Decoupled Architecture) [DO95] that combines simultaneous multithreading with a decoupled architecture.

The MIT J-Machine uses the MDP (Message-Driven Processor) [DFK+92] that consists of a processor, router, as well as internal and external memory. Each memory location is tagged by a four-bit value to support data types and synchronization. Threads are created for every message that arrives in the processor via the router.

The four members of the RS64 processor family [BEKK00][SAB+98] are based on the PowerPC architecture and are one of the first commercial microprocessors using multithreading. The processors are optimized for commercial workloads, e.g. data bases, and support up to two threads. Context is switched on misses to the first level cache, i.e. a dynamic approach.

Simultaneous Multithreading. Simultaneous Multithreading is a combination of the cycle-by-cycle and block interleaving models described above with super-scalar instruction issue. As illustrated in Figure 1.13, a processor that uses simultaneous multithreading issues multiple instructions from multiple threads in each cycle. Simultaneous multithreading combines vertical instruction issue, i.e. filling unused cycles with instructions from other threads, and horizontal instruction issue, i.e. filling unused instruction slots in a cycle with instruction from other threads.

Simultaneous multithreaded processors can be implemented on top of conventional super-scalar processors without adding too much complexity by sharing resources between threads: First, the register file has to be enlarged to support the additional register contexts. Second, the instruction fetch and retire stages have to be changed in order to support fetch/retire of instructions that belong to different threads. Finally, each instruction has to be tagged with a thread specifier inside the pipeline, in order to detect and resolve inter- and intra-thread dependencies. The changes are reported to increase the transistor budget by less than 10 % compared to a conventional super-scalar processor [Die99]. The following paragraphs describe several processors using simultaneous multithreading.

Dynamic Interleaving [PW91] is one of the first approaches to simultaneous multithreading. Simulations based on a conventional load/store RISC architecture with eight function units and support for up to four threads, yielded speedups of 2.5 for the Livermoore Loops and several other programs.

The MRLP (Media Research Laboratory Processor) [HKN+92] is one of the first approaches to simultaneous multithreading. The processor was based on a load/store RISC architecture, extended by dedicated register files and dispatch units for every thread. In addition, inter-thread synchronization is supported by means of queue registers that can be used to directly pass data from one thread to another. Simulations based on configurations of the architecture with eight threads, six execution units, as well as one or two load/store units, yielded speedups of 3.22 and 5.79 for a ray-tracing application, respectively.

The SMT (Simultaneous Multithreading) processor from the University of Washington [EEL+97] is based on an out-of-order super-scalar processor architecture. The processor consists of six integers execution units, three floating-point execution units and supports up to eight threads. There are dedicated program counters and return-address stacks for each thread, all other resources are shared between threads. Thread identifiers are used to distinguish instructions that belong to different threads. Based on simulations, a

speedup of 2.5 on the SPEC92 benchmark suite was reported [TEE+95]. An earlier version of the architecture was reported to achieve four-fold speedups, but used an unrealistic number (32) of function units [TEL95].

The Karlsruhe Simultaneous Multithreading processor [vRU99] is a simultaneous multithreading processor based on the PowerPC architecture, although a simplified instruction set is used. The architecture of the processor is designed to be scalable, i.e. does not limit the number of threads and execution units as well as the size of the register sets and caches. Simulations based on a processor with four threads, 64 renaming registers and two 8 KB first-level caches on a multithreaded workload yielded an IPC of 4.2.

The Alpha 21464 [Die99] processor supports four threads, eight execution units as well as out-of-order issue and efficient thread synchronization. Unlike the other processors described in this section, the 21464 was a commercial product that was expected in the 2003/2004 time frame. However, all work on the 21464 was canceled after the Alpha architecture was discontinued.

The SMV (Simultaneous Multithreaded Vector) [EV97] processor combines simultaneous multithreading with vector instructions, i.e. instructions that operate on a large number of scalars in parallel. The proposed processor supports up to eight threads and consists of an integer and floating-point register file with 128 registers each, as well as four integer and two floating-point execution units. The vector consists of a register file with 128 vector registers and four vector executions units, supporting vector lengths up to 128 elements.

The SDSP (Superscalar Digital Signal Processor) [LB96] [GB96] combines simultaneous multithreading with digital signal processing, emphasizing minimal hardware overhead: Adding simultaneous multithreading only required changes in the instruction fetch and commit stages of the underlying digital signal processor. Based on simulations, speedups between 20 % and 50 % are reported by using up to four threads.

The MCMS (Multiple Context Multithreaded Superscalar) [LW00] processor architecture is similar to the SMT architecture from the University of Washington, but differs in implementation details, most notably synchronization between threads. The architecture uses separate register files and reorder buffers, instruction fetch is restricted to a single thread in each cycle. Based on simulations, a four-thread MCMS architecture achieves a speedup of 1.6 on the SPLASH benchmarks.

The MDA (Multithreaded Decoupled Architecture) [PG99] processor architecture combines simultaneous multithreading with a decoupled architecture, i.e. separates execution and load/store units. The simulated processor consists of four execution and four load/store execution units and supports up to eight threads. There are dedicated fetch and dispatch stages, register files and instruction and address queues for each thread, only the issue logic, execution units and caches are shared between threads.

Summary. Most of the multithreaded processor architectures presented above are research prototypes that have never been implemented or commercialized. Up to now, the only commercial multithreaded processors that are the Tera MTA and the IBM RS64 processor family: The Tera MTA is a supercomputer implemented in Gallium-Arsenide with a single installed system. The RS64 family of processors supports only two threads and is optimized for commercial applications. In addition, these processors are available only in the form of the IBM pSeries server, i.e. are not commercial off-the-shelf products. As the majority of systems still uses conventional single-threaded processors, software multithreading as described in the next section are widely used.

1.4.2 Software Multithreading

Due to the numerous tradeoffs between overhead and functionality, the design space of software multithreading is rather large. Early multithreaded systems were implemented in the operating system kernel, e.g. Mach [ABB+86]. User-level implementations are the current state of the art, since moving from user to kernel space and vice versa significantly increases the context switch overhead. Note that some operating systems, e.g. Solaris, use a combination of kernel- and user-level threads, where several user-level threads are executed on the same kernel thread [SG98].

In contrast to hardware implementations, the number and type of the individual threads in software implementations differ significantly: The number of threads supported by software multithreaded systems is usually unlimited, i.e. only limited by resource constraints. However, the overhead associated with thread management usually increases with the number of threads, hence the feasible number of threads is limited. Based on type, the threads can be grouped into three classes: lightweight threads, very lightweight threads and run-to-completion threads. Lightweight and very lightweight threads are general in the sense that their capabilities are similar to traditional processes, with the exception of shared resources, e.g. address space, file descriptors. Run-to-completion threads can not block and can not be preempted, i.e. context switches are statically incorporated into the thread. However, these restrictions significantly reduce the overhead associated with run-to-completion threads. Note that lightweight and very lightweight threads stem from research in operating systems, while run-to-completion threads stem from research in programming languages.

The most notable difference between software multithreading systems is the type of address space supported by the thread system, i.e. shared or distributed. In the former case, all threads share the same address space, while some threads may reside in a different address space, i.e. on a different processor, in the latter case. Both alternatives are surveyed in the following sections.

Shared Address Space. C Threads [CD88] is a user-level thread package that enables parallel programming using the C language on the MACH operating system. The library can be built on top of coroutines, threads, or processes, depending on compile-time options. The programming model supports forking and joining of threads as well as mutexes and condition variables for synchronization.

PCR Threads [WDH89] is a user-level thread package that distributes the individual threads to multiple processes, instead of a single process as used by most other thread packages. The advantage of this approach is that only a subset of the threads are blocked whenever a process is blocked, i.e. those threads that were assigned to the corresponding process. Note that the processes share the same address space, hence the operating system must support this feature.

First-Class User-Level Threads [MSLM91] are part of the Psyche operating system that supports efficient execution of user-level threads. This is achieved by providing a shared memory for asynchronous communication between user- and kernel-level and a scheduling strategy that allows synchronization of user- and kernel-level scheduling. Note that Psyche is not restricted to a specific user-level thread package, several such packages have been modified to take advantage of these features.

Osprey [PE96] is a very lightweight thread package that increases efficiency by separating the specification of a thread from its context. As the corresponding data structure is much smaller than the context of a thread, these threads can be efficiently managed, especially if the number of threads is large. Note that the threads in Osprey are general, i.e. are not restricted in any way.

POSIX Threads [But97] is a standardized thread package that is similar to C threads described above. Although POSIX threads are usually implemented as a user-level library [Mue93], there are some kernel-level implementations [Alf94].

Presto [BLL88] is a programming system for object-oriented parallel programming based on the C++ [Str97] language. The corresponding runtime system supports threads as objects for parallel execution as well as locks and condition variables for synchronization.

AWESIME (A Widely Extensible Simulation Environment) [Gru91] is an object-oriented library for parallel programming based on the C++ language, similar to Presto described above. The notable difference is the support for global time ordering for the execution of threads, which is useful for simulations, e.g. circuit simulations.

The Chorus [EZ93] system is a runtime system that supports the chore programming model. A chore consists of a collection of atoms, i.e. sequential tasks that represent multiple applications of the same function. Execution ordering between atoms in the same chore is supported. The runtime system

uses several Presto threads per processor, that execute the individual chores from a work-heap. Note that only one of the threads is active at any time.

Filaments [LFA96] is a user-level thread package that supports different types of threads, i.e. run-to-completion threads, iterative and fork/join threads. Filaments supports stateless threads without a private stack, small thread descriptors, and scheduling for data locality.

The TAM (Threaded Abstract Machine) [CSS+91] is a virtual machine that supports efficient management of processor resources and thread scheduling. The virtual machine is the target of a parallelizing compiler, the individual threads are run-to-completion.

Fast Threads [ALL89] is a user-level thread package that was used to evaluate several thread management and locking alternatives. The package itself is similar to Presto, but is claimed to be an order of magnitude faster. The Fast Threads package was later modified to run on top of scheduler activations [ABLL92] instead of kernel-level threads. Scheduler activations is a technique for efficient scheduling of user-level threads, similar to approach in the Psyche operating system.

A user-level thread package that supports run-to-completion threads and scheduling for cache locality is described in [PEA+96]. The scheduling strategy uses several address hints associated with each thread and computes an execution ordering that minimizes thread misses.

The Uniform system [TC88] is a runtime system for shared memory machines that uses a one-to-one mapping of processes to processors to execute individual threads. These threads are created by task generators, i.e. several threads are created at once and are restricted to the run-to-completion model.

Distributed Address Space. NewThreads [FM92] is a user-level runtime system for distributed memory machines that supports multiple threads per processor. The thread subsystem is similar to scheduler activations described above, threads on the same or different processors communicate via message-passing. Note that these operations block the corresponding thread.

Chant [HCM94] is a user-level thread package for distributed memory machines that is targeted at latency tolerance. The individual threads communicate via port-to-port or remote procedure calls, the programming interface is an extension of POSIX threads with added support to distinguish threads on different processors.

Rthreads [DZU98b][DZU98a] is a software distributed shared memory system based on multithreading. Rthreads provides primitives that are similar to the primitives provided by POSIX threads, such that programs written for POSIX threads can be transformed automatically into programs using Rthreads. In addition, both multithreading systems can be mixed in the same program. In contrast to POSIX threads, the Rthreads system supports distributed memory, e.g. a cluster of workstations. All global variables are shared automatically, pointers to such variables are not supported. The

Rthreads system consists of a precompiler that transforms the source code of programs using POSIX threads, and a small library that implements the individual primitives. Rthreads primitives support consistency for synchronization primitives, there is no restriction on the ordering of normal accesses. In order to ensure sequential consistency, read and write accesses to global variables have to be performed using explicit primitives. These primitives are buffered until a subsequent flush primitive is encountered, afterwards the affected data is transfered in a single operation. The Rthreads system has been implemented on a cluster of workstations running the AIX operating system. Experimental results show that the Rthreads system generates only a slight overhead compared to the PVM system and is superior to the Adsmith distributed shared memory systems.

UPVM [KCO+94] is a user-level thread package targeted at PVM (Parallel Virtual Machine) applications. Each thread uses private data and heap spaces in contrast to other user-level thread packages. The programming model is therefore similar to traditional processes, but in contrast to processes, these data and heap spaces are not protected from the other threads.

Distributed Filaments [FLA94] is an extension of filaments that is targeted at networks of workstations. The distributed filaments packages consists of a multithreaded distributed shared memory system, as well as communication and synchronization primitives.

Cilk [BJK+96] is a runtime system for parallel programming on distributed memory machines and networks of workstations based on the C language. A Cilk program is a directed acyclic graph, where the nodes represent the individual threads and a set of threads comprises a procedure. The individual threads are not general, i.e. use the run-to-completion model. Cilk uses a work-stealing approach, i.e. supports migration of threads between processors.

VISA [HB93] is a runtime system for functional languages on distributed memory machines that employs multithreading to tolerate the latency of remote references. The runtime system uses the standard setjmp() and longjpm() routines to implement context switches, hence threads have to be resumed in reverse order.

Virtual Processors [NS97] is a runtime system that provides a virtual machine abstraction for fine-grained parallel code. The runtime system is based on multithreading with very lightweight threads. The approach uses communication-based scheduling, tight integration between communication and scheduling, small thread descriptors, and a zero-copy strategy for internal communication.

1.4.3 Summary

Although a large number of software multithreading systems exits, none was specifically targeted at latency tolerance in massively parallel processors. As

these machines usually do not support a shared memory model, such a multithreading system must support a distributed memory model. Unfortunately, most software multithreading systems support only the shared memory system. Although there are several multithreading systems that support a distributed memory model, these systems were not targeted at latency tolerance and do not support the corresponding small grainsizes. Chapter 2 describes the basic concept of emulated multithreading, a software multithreading system that supports sufficiently small grain sizes to enable latency tolerance in massively parallel processors.

1.5 Outline

The remainder of this work consists of six chapters and two appendices that are organized as follows: Chapter 2 derives the basic features of a software multithreading system that is designed to enable latency tolerance for remote memory accesses in massively parallel processors. This approach is called emulated multithreading. The current implementation of emulated multithreading, especially the algorithms used to implement this technique, are described in Chapter 3.

Emulated multithreading is evaluated on two different platforms: The Compaq XP1000 workstation, a single-processor system based on the Alpha architecture, as well as the Cray T3E, a massively parallel processor that is based on the Alpha architecture. The choice of these systems was influenced by the properties of the Alpha architecture, which is well suited for emulated multithreading for reasons which are outlined in Section 2.4. A description of these platforms, as well as results of the evaluation on a set of benchmarks are covered in Chapters 5 and 6, respectively. The evaluation uses six different benchmarks from a popular parallel benchmark suite, the choice of benchmarks as well as a detailed description of the individual benchmarks can be found in Chapter 4. Chapter 7 summarizes the individual chapters as well as the results of the evaluation on both platforms and provides an outlook into the future directions of emulated multithreading. The Alpha architecture as well as implementations of this architecture are described in Appendix A. Appendix B contains a very detailed programming guide to the E-registers of the Cray T3E and includes information that is not publicly available elsewhere.

2. Emulated Multithreading

The previous chapter has identified the latency of accesses to remote memory as a major bottleneck in current massively parallel processors. Emulated multithreading is designed to tolerate this latency, thereby increasing performance. This goal is achieved by using a combination of software multithreading and split-transaction communication routines on massively parallel processors. The current chapter covers the implementation-independent aspects of emulated multithreading, while Chapter 3 describes the implementation-specific issues.

Section 2.1 explains the rationale behind the preferences for the fundamental design, the basic concept of emulated multithreading is described in Section 2.2 based on these preferences. The interaction between emulated multithreading and the underlying processor architecture is described in Section 2.3. Section 2.4 surveys current processor architectures and evaluates each architecture with respect to emulated multithreading.

2.1 Design Preferences

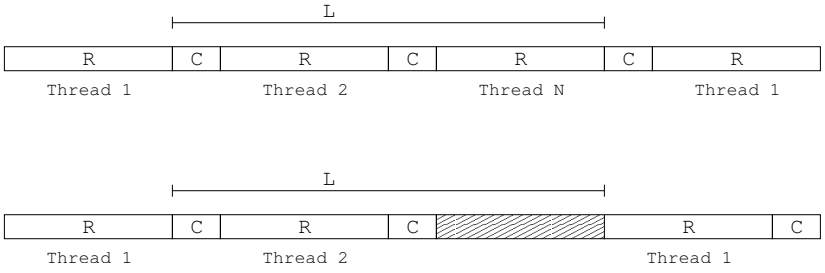
This section explains the rationale behind the preferences for the design of emulated multithreading. A simple model for multithreaded processors is used to explain the individual design choices. This well-known model is introduced in Section 2.1.1. Sections 2.1.2 and 2.1.3 cover the context-switch strategy and the techniques used to reduce context switch overhead, respectively.

2.1.1 Multithreaded Processor Model

A simple model for multithreaded processors is used to explain the individual preferences for the design. This model is widely used in literature to explain the basic issues of multithreaded processors. The model is not accurate enough to allow detailed performance predictions, other models provide much more detail and are better suited for this task, e.g. [SBCvE90][Aga92][BL96] [VA00]. However, the model is detailed enough for the following discussion regarding the fundamental design choices for emulated multithreading.

The model has four parameters, which are supposed to be constant:

Fig. 2.1. Processor Utilization using a Model of Multithreading



N The number of threads that are executed by the processor.
R The run-length, i.e. the number of cycles between long-latency events.
L The duration of the long-latency events in cycles.
C The number of cycles required to switch to another thread.

The model assumes that a thread performs useful work for R cycles, then encounters a long-latency event, and switches execution to another thread, i.e. it uses a block-interleaved approach.

The processor utilization u, i.e. the ratio of useful work performed in reference to total execution time, can be determined by distinguishing two different cases as described in the following paragraphs.

In the first case, depicted in the upper part of Figure 2.1, the number of threads N is large enough to hide the long-latency event completely. In this case, the processor utilization u is given by

$$u = \frac{R}{R+C} = \frac{1}{1+\frac{C}{R}} \quad . \tag{2.1}$$

The number of threads required to completely hide the long-latency events can be determined as follows: The run-length of $N-1$ threads and N context switches must be larger than the duration of the long-latency event, i.e.

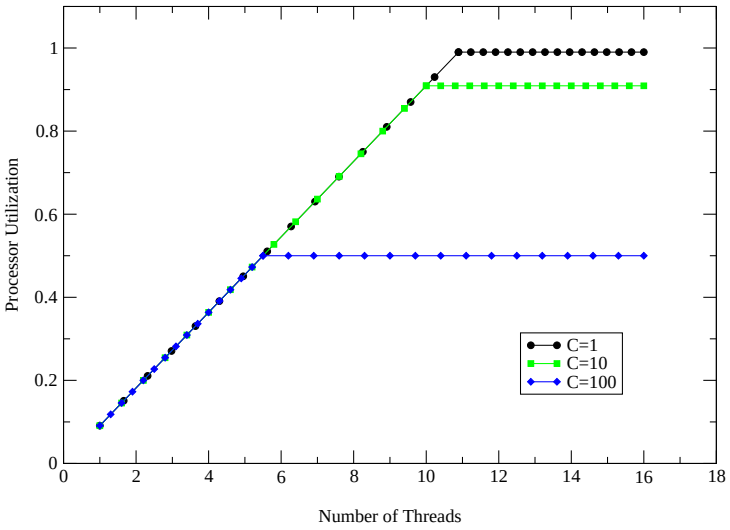
$$(N-1)R + NC \geq L \Leftrightarrow N = \frac{R+L}{R+C} \quad . \tag{2.2}$$

Note that equation 2.1 depends only on R and C, i.e. the context switch time C governs the upper bound on processor utilization, since the run-length R is application-dependent.

In the second case, depicted in the lower part of Figure 2.1, the number of threads is too small to hide the long-latency events completely. In this case, the processor utilization u is given by

$$u = \frac{NR}{R+L} = \frac{1}{1+\frac{L}{R}} \cdot N \quad . \tag{2.3}$$

Fig. 2.2. Processor Utilization for R=100 and L=1000



since N threads perform useful work for R cycles each during a total of $R+L$ cycles. Note that equation 2.3 is linear in the number of threads N.

The impact of context switch time on processor utilization is depicted in Fig. 2.2. This figure uses the equations derived above and assumes that R=100, L=1000, and C=1, 10, 100. Since the upper bound on processor utilization is governed by context switch time, the reduction of this time is essential for achieving good performance. As emulated multithreading is implemented in software, reducing the context switch time without additional hardware support is one of the key challenges. Another observation is the need for split-transaction communications, i.e. the ability to perform other work after initializing a remote access. Unfortunately, the implementation of such protocols is not straightforward in current massively parallel processors.

2.1.2 Context Switch Strategies

The model of multithreaded processors which has been introduced in the previous section assumes that context is switched to another thread after a long-latency event is encountered, thereby trying to tolerate the latency of the event. Emulated multithreading is targeted at tolerating latencies in massively parallel processors, hence the model corresponds with this intention of emulated multithreading. The model makes no assumptions about the type of events and the scheduling strategy. Therefore two decisions have to be made regarding the context switch strategy: The type of events that cause a context switch as well as the scheduling strategy. The next paragraphs address these questions.

The selected events should have two properties: First, the latency of the event should be at least two times larger than the context switch overhead. Otherwise it is impossible to tolerate the latency of the event with useful work, since the context switch itself is not considered useful work. Second, the occurrence of the event should be identifiable in an automated fashion with reasonable accuracy. Otherwise the context switch is performed in the absence of the event, i.e. the total overhead is increased although no benefits are possible.

The following long-latency events arise in current microprocessors and massively parallel processors: complex instructions, misses at different levels of the local memory hierarchy, and remote memory accesses. Another type of long-latency events, i.e. I/O requests, is not considered, since these events are usually handled by calls to the operating system. These calls are atomic in most cases, i.e. it is not possible to split initiation and completion of I/O requests, which is a prerequisite for latency tolerance.

Complex instructions are easily identified by inspecting the assembler source, but the latency of these instructions is on the order of tens of cycles. Since emulated multithreading is implemented in software, it is unlikely that the context switch overhead is small enough to make switching on these events feasible. The same reasoning holds for cache misses that hit in other caches: These misses have a latency on the order of tens of cycles. Memory accesses that miss in all caches, i.e. access local memory, are more interesting due to their higher latency on the order of hundreds of cycles. However, it is very hard to determine in advance, whether a given memory access instruction will miss in all caches. Profiling information gathered from previous runs of the program can provide hints in this regard, but requires a tight integration with the compiler. The use of profiling information was estimated to be too complex for the initial implementation of emulated multithreading. Therefore the initial implementation does not switch on these events. This capability can be retro-fitted once the necessary level of compiler integration is achieved. Section 3.8 discusses this and other benefits of tighter compiler integration.

Apart from profiling, informing memory instructions are able to identify memory accesses, that miss in all caches at run time [HmMS98]. These instructions perform an additional operation, e.g. a branch, if the memory access misses in specified levels of the memory hierarchy. As already mentioned in the previous chapter, a software multithreaded system based on informing memory instructions is feasible and leads to encouraging results: Even if a full context switch is performed on every miss in the second-level cache, the simulated results show performance improvements on the order of 10 % to 23 % for several benchmarks [MR99]. Unfortunately, informing memory instructions have been implemented neither in commercial microprocessors nor in research prototypes.

In the case of massively parallel processors, accesses to remote memory incur a latency on the order of thousands of cycles. Since these accesses are

usually performed using some form of communication library, e.g. shmem [Cra98b], MPI [SOHL+98][GHLL+98], or PVM [GBD+94], these accesses can be identified easily by inspecting the assembler code and looking for calls to certain library routines. Note that some of these routines will access local memory, but the majority of accesses will be to remote memory: If every operation is guaranteed to access only local memory, the corresponding call should be replaced by a normal load or store instruction, i.e. the application should be changed.

In conclusion, emulated multithreading switches contexts on accesses to remote memory. Future implementations might use profiling to enable switching on cache misses as well. As the latency of remote memory accesses is on the order of thousands of cycles, the context switch overhead might seem insignificant. However, it will be necessary to perform context switches even in the absence of long-latency events, as will be shown later. The context switch overhead is therefore important and should be kept small as much as possible.

2.1.3 Context Switch Overhead

The overhead associated with a context switch is important for the overall performance of emulated multithreading. This section derives ways to reduce the context switch overhead. This is accomplished by looking at the different elements of a context switch and optimizing each element for performance. A context switch consists of storing the context of the current thread to memory, scheduling the next thread and restoring the context of that thread from memory. The next paragraph derives a suitable scheduling strategy as well as optimizations of the save and restore operations of thread contexts.

There are many different scheduling algorithms, an introduction can be found in [SG98]. The most important scheduling strategies are:

- First-Come, First-Served (FCFS) scheduling is the simplest scheduling algorithm: The first thread that requests the processor is executed. This scheduling strategy is non-preemptive, i.e. the currently executing thread can not be interrupted, but has to release the processor voluntarily instead. First-come, First-Served scheduling is easily implemented using a first-in, first-out queue.
- Round-Robin (RR) scheduling is similar to FCFS scheduling, but adds preemptiveness. Every thread is allocated a time slice, context is switched either if the thread releases the processor voluntarily, or if the corresponding time slice has elapsed. This scheduling strategy can be implemented with a first-in, first-out queue and timer interrupts.
- Shortest-Job-First (SJF) scheduling assigns the processor to the thread with the shortest request for processor time. This scheduling algorithm is proven to be optimal [SG98], but the length of the individual processor requests is usually unknown in advance and has to be estimated.

Shortest-Job-First scheduling can be implemented in a preemptive or non-preemptive fashion.

- Priority scheduling assigns a priority to each thread and allocates the processor to the thread with the highest priority. First-come, first-served scheduling is used for threads that have the same priority. Priority scheduling can be implemented in a preemptive and a non-preemptive fashion.

In order to reduce the context switch overhead, the simplest scheduling algorithm, i.e. first-come, first-served scheduling, is chosen for emulated multithreading. This algorithm is easily implemented by using a circular queue: The scheduling algorithm simply traverses the queue until all threads have finished execution. Note that response time is no criterion for the purpose of tolerating long-latency events: Even if a thread uses the processor for a prolonged period of time, the thread still performs useful work. The larger the time spent performing useful work, the more latency for the other threads can be tolerated. Preemptive scheduling is therefore not necessary for emulated multithreading. Hence first-come, first-served scheduling was chosen over round-robin scheduling. The other scheduling algorithms are too complex for the purposes of emulated multithreading, where the running time of the scheduling algorithm is the most important characteristic.

As already mentioned above, a context switch needs to store the context of the current thread and restore the context of the next thread. The context of a thread consists of the registers that are defined by the underlying processor architecture, as well as some management information. Since current RISC architectures usually define 64 general-purpose as well as some special registers, the size of the context is dominated by the size of the register set. The latency of the save and restore operation increases with the size of the context, since every register has to be saved or restored by a separate load or store instruction, respectively.

The number of general-purpose and special registers is specified by the underlying processor architecture and cannot be changed, therefore it is impossible to reduce the size of the context. However, it is usually not necessary to save and restore the whole context of a thread. Instead only the necessary parts of the context have to be saved and restored [Wal86][LH86][GN96]. For example, programs that perform mostly integer operations will seldom use any of the floating-point registers, thereby providing ample opportunity to reduce the number of save and restore instructions.

In order to exploit this opportunity, two problems have to be solved: The first problem is the identification of those registers, that are in use at a given program location. The second problem is the context switch routine itself, which must be able to save and restore only those registers, that have previously been identified to be in use.

Recall that emulated multithreading uses non-preemptive scheduling, i.e. a thread executes until it voluntarily releases the processor. Regarding the first problem, the identification of such registers can be restricted to specific

locations. Note that the number and location of used registers is already known at compile-time: The compiler gathers this information via data-flow algorithms prior to register allocation. It is therefore possible to determine the number and location of the used registers at compile-time and transfer this information to the context switch routine, e.g. via a register mask. The context switch then uses the register mask to determine which registers have to be saved and restored. However, a generic context switch routine that handles all possible cases would be fairly complex, e.g. contain lots of branches, thereby consuming some or all of the saved overhead.

In order to keep the context switch routine simple, it is possible to use several context switch routines that are tailored to each context switch location. The tailored context switch routines have to be called at the start and end of each instruction block, hence corresponding calls have to be inserted at these locations. This approach allows the individual context switch routines to be tailored to the individual context switches, at the expense of code size and an increased number of calls. Note that the last problem can be solved by integrating the tailored context switch routine in the program code. One the one hand, this approach further increases code size as the reuse of the context switch routines is no longer possible. On the other hand, this approach no longer requires expensive calls at the start and end of each instruction block. After merging the program code with the tailored context switch routines, even further optimizations are possible by spreading the context switch code along the instruction block, possibly filling empty instruction slots.

Due to the large numbers of registers in current RISC architectures, it is possible to partition the register set between a small number of threads [WW93]. If each thread uses only the registers in the corresponding partition, the save and restore of these registers upon context switch can be omitted. The major drawbacks of this scheme are the increased number of register spills due to the smaller number of registers in each partition, and calls to routines that do not use emulated multithreading. These routines adhere to the standard calling conventions and can therefore not be restricted to one of the partitions, hence all registers have to be saved before and restored after the call, respectively. Otherwise the routine would destroy the contents of registers that belong to another partition, i.e. another thread. Note that every thread has to execute different versions of the program tailored to the corresponding partition, thus further increasing code size. Since register partitioning has some drawbacks and allows only a small number of threads, it will be optional for emulated multithreading.

2.2 Basic Concept

The previous section explains the preferences for the design of emulated multithreading. Based on these preferences, this section covers the basic concept of emulated multithreading. Section 2.2.1 describes the assumptions made

about programs that are transformed to use emulated multithreading. Section 2.2.2 discusses the data structures that are used by emulated multithreading, while Section 2.2.4 describes the conversion process that merges application and context switch code. The functionality of the emulation library that supports emulated multithreading is covered in Section 2.2.3.

2.2.1 Assumptions

In order to utilize emulated multithreading, several assumptions are made: about the programs: First, the programs must already support multithreading, e.g. by using POSIX threads. In the case of parallel programs, all accesses to remote memory and synchronization operations must use explicit shared memory primitives, an extension to message-passing primitives is not yet implemented. As the emulation library provides primitives that are similar to the primitives provided by POSIX threads, as well as shared memory primitives, programs that meet these conditions can be automatically converted to use emulated multithreading, e.g. by using a precompiler. Such a precompiler is feasible, as similar programs already exists, e.g. as part of the Rthreads package [DZU98b]. Second, the high-level language source code of the converted programs is assumed to be available. The conversion process is based on the assembler sources that are generated from the high-level language sources by the compiler. Note that it is possible to perform the modifications described above on object files or executables, e.g. by using the EEL library [LS95] or the OM [SW93] and ATOM [SE94] tools. However, such an approach would unnecessarily add complexity to the conversion process. Second, the converted program must not be self-modifying, otherwise the code conversion is likely to break the semantics of the program. However, this is not a serious restriction, since self-modifying code is generally considered as unfavorable.

2.2.2 Data Structures

Emulated multithreading uses two major data structures: the thread control block and the thread descriptor. The thread control block is used to manage a group of threads, while the thread descriptor contains the context of a thread as well as some management information.

The thread control block contains the following elements:

- The ActiveHead and ActiveTail pointers contain the addresses of the thread descriptors at the head and tail of the active thread list, respectively. The active thread list is a circular, doubly-linked list that contains the thread descriptors for all threads that are currently executed.
- The IdleHead and IdleTail pointers contain the addresses of the thread descriptors at the head and tail of the inactive thread list, respectively. The inactive thread list contains the thread descriptors for all threads that are currently suspended, i.e. during a barrier synchronization.

The thread descriptor contains the following elements:

- The NextThread and PreviousThread pointers are used to insert the thread descriptor into a doubly-linked list of thread descriptors.
- The ThreadCtrl pointer contains the address of the thread control block that manages the corresponding group of threads.
- The Number word contains the unique number that is assigned to each thread.
- The Status word contains the current call-depth of the thread.
- The PC pointer contains the address of the instruction block that is to be executed upon the next invocation of the thread.
- The IReg array stores the contents of the general-purpose integer registers, zero source and sink registers are not stored explicitly.
- The FReg array stores the contents of the general-purpose floating-point registers, zero source and sink registers are not stored explicitly.

The elements described above are mandatory for emulated multithreading. Implementations may need to add other implementation-specific elements to the thread descriptor data structure. The size s of the thread descriptor is given by

$$s = 4 \cdot \text{sizeof(long)} + 2 \cdot \text{sizeof(int)} + n_{\text{int}} \cdot \text{sizeof(long)} + n_{\text{fp}} \cdot \text{sizeof(double)}$$

where n_{int} and n_{fp} are the number of general-purpose integer and floating-point registers, respectively, minus the corresponding number of zero source and sink registers. For example, the Alpha architecture defines 32 integer as well as 32 floating-point registers and uses one zero source and sink register for each register type, i.e $n_{\text{int}} = n_{\text{fp}} = 31$. Therefore the thread descriptor for the Alpha architecture consumes 536 bytes. Note that the data structure may be padded in order to align the individual thread descriptors on cache-line boundaries to avoid conflict misses.

Apart from the two data structures described above, emulated multithreading uses several other data structures: The thread attribute data structure is used to specify attributes for a group of threads. The thread arguments structure is used to specify the initial arguments for a group of threads. Both data structures are passed to the thread creation routine.

2.2.3 Emulation Library

Emulated multithreading uses a small support library that is linked to the converted application. This emulation library contains routines for thread initialization, thread execution as well as communication and synchronization routines. Thread initialization routines are used to initialize and manage the thread arguments and thread attribute data structures as well as creating the individual threads. The thread creation routine takes as arguments the thread attribute and arguments structures as well as the entry point for the

threads. It creates and initializes a thread control block as well as thread descriptors for all threads. The returned thread control block is subsequently passed to the thread execution routine to start the execution of threads.

The thread execution routine executes the individual threads using first-come, first-served scheduling as described in Section 2.1.3. The routine basically consists of a single loop and operates in the following way:

1. Upon entry, the FramePtr register is set to the first thread in the circular list of active threads specified by the thread control block.
2. The program counter is loaded from the current thread descriptor and stored in the ThreadPC register.
3. A call to the address given by the ThreadPC register is performed to execute the corresponding instruction block. The call sets the ReturnPC register to the next instruction, i.e. the fourth step.
4. After the instruction block has finished execution, it returns to the address given by the ReturnPC register, and updates the ThreadPC register accordingly. The ReturnPC register is subsequently saved to the current thread descriptor.
5. The status of the current thread is inspected. If the current call depth is zero, i.e. the thread has completed, the thread descriptor is removed from the active list.
6. The FramePtr is loaded from the NextThread field of the current thread descriptor, i.e. points to the next thread. If the updated FramePtr is non-zero, return to step two.

The thread execution routine consists of a single loop and uses three registers: the ThreadPC, ReturnPC and FramePtr registers, respectively. Since the loop is iterated once for every context switch, it should be compact and fast. Fortunately, this loop can be implemented in less than 10 instructions for most RISC architectures. Note that Section 2.3 includes a discussion of performance issues regarding the thread execution routine.

The communication routines are largely platform-specific and provide split-transaction memory accesses. The synchronization routines are platform-specific as well and provide barriers and locks. These routines are described in Chapter 3.

2.2.4 Code Conversion

As already mentioned, emulated multithreading divides the application code into instruction blocks. These blocks are merged with context switch code, that is tailored to each instruction block, and transformed into subroutines. This section describes the conversion process in more detail. Note that the following remarks make no assumptions about the size and format of the instruction block. The process of selecting instruction blocks is implementation-specific and therefore covered in Chapter 3. The following paragraphs describe the general conversion process as well as some important special cases.

Given an instruction block B, which is assumed to be identical to a basic block for the sake of the following discussion, the block is transformed as follows: For each instruction I in the current block, the registers that are used by I are identified. If Instruction I is the first instruction in the current instruction block to use a register R, allocate a new register R', otherwise use the previously allocated register R'. In addition, if instruction I reads register R, insert a restore instruction before instruction I. The restore instruction loads register R' with the contents of register R as stored in the descriptor of the current thread. The restore instruction can be omitted if register R is only written by instruction I, as the loaded value would be immediately discarded by executing instruction I.

If instruction I is the last instruction in the current instruction block to use a register R, a save instruction is inserted behind instruction I and register R' that was previously allocated to register R is freed. The save instruction stores the contents of register R' to register R in the descriptor of the current thread. The save instruction can be omitted if register R' is not modified by any of the instructions in the current instruction block.

After parsing all instructions and modifying the individual instructions according to the rules above, a return instruction is appended to the instruction block. This return instruction is used to return execution to the thread execution routine after execution of the instruction block has completed.

The conversion process is illustrated by the following example, an instruction block that consists of three instructions. The example is taken from a RISC architecture with a three-address instruction format: The first two register operands specify the source registers, the last register operand specifies the target register.

```
addq    r1, r2, r3
subq    r4, r5, r6
addq    r3, r6, r3
```

This instruction block is converted as follows: The first addq instruction reads source registers r1 and r2 and writes the result to destination register r3. As the addq instruction is the first instruction in the instruction block, all three registers are used for the first time and are allocated to registers r1', r2' and r3'. Since registers r1 and r2 are read by the addq instruction, two restore instructions are inserted before the addq instruction. Register r3 does not need to be restored, as it is initialized by the addq instruction. The first addq instruction is the last instruction to use registers r1 and r2, but those registers are not modified by any of the instructions in the instruction block, hence no save instructions need to be inserted and registers r1' and r2' are freed. Register r3' does not need to be saved, since it is used later by other instructions in the instruction block.

The subq instruction reads source registers r4 and r5 and writes the result to destination register r6. Although the subq instruction is not the first instruction in the instruction block, all three registers are used for the first time

and are allocated to registers r1', r2' and r4'. Since registers r4 and r5 are read by the addq instruction, two restore instructions are inserted before the subq instruction. Register r6 does not need to be restored, as it is initialized by the addq instruction. The subq instruction is the last instruction to use registers r4, r5, but those registers are not modified by any of the instructions in the instruction block, hence no save instructions have to be inserted and the registers r1' and r2' are freed. Register r6 does not need to be saved, since it is used later by other instructions in the instruction block.

The second addq instruction reads registers r3 and r6 and writes the result to register r3. All registers used by this instruction have been used by previous instructions, hence there is no need to insert any restore operations or allocate any registers. However, the second addq instruction is the last instruction in the instruction block and contains the last reference to registers r3 and r6. As both registers have been modified by this or other instructions in the instruction block, two save instructions have to be inserted after the second addq instruction and the registers r3' and r4' are freed afterwards. The return instruction is appended to the instruction block.

The converted instruction block is given below. The `ldq Rx. #0(Ry)` instructions load the contents of the memory location given by `Ry + 0` into the destination register `Rx`.

```
ldq r1', #Ireg[r1](FramePtr)
ldq r2', #Ireg[r2](FramePtr)
addq    r1', r2', r3'
ldq r1', #Ireg[r4](FramePtr)
ldq r2', #Ireg[r5](FramePtr)
subq    r1', r2', r4'
addq    r3', r4', r3'
stq r3', #Ireg[r3](FramePtr)
stq r4', #Ireg[r6](FramePtr)
ret ThreadPC, (ReturnPC)
```

The above description of the conversion process covers only non-control-flow instructions, i.e. instructions that do not change the flow of the program. There are three different types of control flow instructions: conditional, unconditional, and indirect branches. Recall that instruction blocks are identical to basic blocks for the sake of this discussion. As branches end basic blocks, a branch is always the last instruction in an instruction block. Hence branches are replaced by a sequence of instructions that calculate the address of the updated program counter, store this address in the ReturnPC register and return to the thread execution routine. For example, the conditional branch `beq r1, target` that performs a branch if register r1 is equal to zero, would be replaced by the following instruction sequence:

```
ldq    r1', #Ireg[r1](FramePtr)
lda    r2'  target
```

```
lda    ThreadPC, next
cmoveq r1', r2', ThreadPC
ret    r31, (ReturnPC)
next:
```

The first ldq instruction restores the contents of register r1 from the thread descriptor to register r1', while the first lda instruction loads the target constant into register r2'. The second lda instruction loads the ThreadPC register with the address of the instruction that follows the original branch instruction, i.e. the instruction marked with the next label. The cmoveq instruction checks whether r1' is equal to zero and moves the contents of register r2' to register ThreadPC in that case, i.e. performs a conditional move. If r1' is equal to zero, the ThreadPC register contains the target constant, the address of the next instruction otherwise. The ret instruction performs an indirect branch to the location given in the ReturnPC register, i.e. the fourth step in the thread execution routine. Recall that the thread execution routine automatically stores the contents of the ReturnPC register to the PC field of the current thread descriptor.

Procedure calls are handled differently, depending on the type of the called procedure. If the called procedure (callee) does not use emulated multithreading, i.e. an external call, the procedure call instruction is treated like a normal instruction and converted accordingly. The only difference to a normal instruction is the register assignment, which has to follow the calling conventions of the operating system. This allows the call to be executed in the normal way, i.e. no modifications to the callee are necessary. As a consequence, emulated multithreading can be applied to selected procedures only, handling other procedures like external calls. Therefore the overhead of emulated multithreading can be reduced by applying emulated multithreading only to those procedures that may benefit from the conversion, e.g. procedures that access remote memory.

If the callee uses emulated multithreading, the call is the last instruction in the instruction stream by definition. Similar to branches, the subroutine call is therefore replaced by a sequence of instructions that store the address of the callee in the ReturnPC register, increment the call depth field of the current thread descriptor, and return to the thread execution routine.

2.3 Performance Issues

As mentioned in the previous section, emulated multithreading requires modifications of the original source code. The possible impact of these modifications on the performance, i.e. the overhead associated with emulated multithreading, is discussed in the current section. The discussion emphasizes the interaction of emulated multithreading with the characteristics of modern mi-

croprocessors and addresses several issues: the number of threads, instruction and data caches, branch prediction, and out-of-order execution.

2.3.1 Number of Threads

Recall that the loop in the thread execution routine can be implemented in less than 10 instructions, which may seem not many at first sight. However, the loop contains at least one conditional branch as well as one subroutine call, two instructions that can impact performance if not predicted correctly. The interaction of emulated multithreading and branch prediction is covered in Section 2.3.3. Nevertheless, in order to reduce the overhead associated with emulated multithreading, it is important to reduce the number of loop iterations as much as possible. The next paragraph derives and analyzes a formula for the number of loop iterations.

A formula for the number of loop iterations can be derived as follows: Assume that the source code was divided into N instructions blocks, numbered from 1 to N, and that block i is executed n_i times by each thread for a given input set. The values for the n_i can be obtained via profiling, for example. Let us further assume that t threads are used and that all n_i values are independent of the number of threads. Based on these assumptions, the number L of loop iterations is

$$L = t \cdot \left(\sum_{i=1}^{N} n_i \right) \quad . \tag{2.4}$$

Note that this equation is simplistic, since the n_i may be different for each thread and usually depend on the number of threads, but it is accurate enough for the sake of the following discussion. Looking at the above equation, three ways to decrease the number of loop traversals are possible:

First of all, reducing the number of threads will reduce the number of loop iterations, since the equation depends linearly on t. However, reducing the number of threads decreases the amount of latency that can be hidden. As a consequence, the number of threads should be as large as needed, but as small as possible.

Second, reducing the number of instruction blocks, e.g. via merging, reduces the number of iterations. Since the number of merged blocks is smaller and none of the merged blocks is executed more often than one of the original instruction blocks, instruction blocks should be as large as possible. However, instruction blocks must end at remote memory accesses in order to hide the latency, hence there is an upper limit on the useful size of instruction blocks.

Third, reducing the number of executions for individual instruction blocks will reduce the number of loop traversals. Like before, this is possible by increasing the size of instruction blocks: If an instruction block is large enough to contain a whole loop, i.e. head, body, and tail, the number of executions for the instruction block is equal to the number of executions of the head and

tail blocks, but independent on the number of executions for the loop body. As a consequence, instruction blocks should be as large as possible, but the restrictions mentioned above still apply.

2.3.2 Caches

Caches reside in the upper levels of the memory hierarchy, i.e. are used to bridge the performance gap between the clock frequency of current microprocessors and the latency of main memory. The importance of caches was already outlined in Chapter 1: Caches are an integral part of current microprocessors, therefore the interaction of emulated multithreading with caches is relevant for the performance of emulated multithreading. The following paragraphs provide a short introduction to caches as well as a discussion of the interactions between caches and emulated multithreading. More detailed information about caches can be found in computer architecture textbooks [HP96][PH98][Fly95][Sto93], or Handy's book [Han93], which is a useful reference to the design of caches.

Cache Functionality. Caches are based on the principles of locality in program behavior: spatial and temporal locality. Spatial locality means that if a program accesses a memory location, it is likely that the program will access nearby locations in the future. Temporal locality means that if a program accesses a memory location, it is likely that the program will access the same location again in the near future.

A cache is a type of memory that resides between the processor and main memory and contains a subset of the contents of main memory, i.e. uses replication. Since caches are usually much smaller than main memory, they can be built from faster memory, thus reducing the latency associated with accesses to memory locations found in the cache. Caches consist of a number of cache-lines, where the size of a cache-line is usually a multiple of the machine word size in order to exploit spatial locality. The size of a cache is given by the number of cache-lines times the number of bytes that can be stored in each cache-line.

Caches exploit the principles of locality mentioned above in the following way: If the processor issues a read request to a location in main memory, the cache checks whether it contains the contents of the desired location. If the contents are found in the cache (cache hit), they are immediately returned to the processor. In the case of a cache miss, the cache-line that contains the desired location is fetched from main memory and stored in the cache for later references. The contents of the desired location are subsequently forwarded to the processor.

If the cache is already full, i.e. each cache-line contains a valid entry, one of the existing entries has to be evicted from the cache if the cache-line can reside in more than one entry. Several strategies to select the entry to be evicted exist. The two most common are random replacement and least-recently-used (LRU). In the case of random replacement, one of the existing

entries is selected pseudo-randomly, which is easy to implement. In the case of least-recently-used replacement, the entry that has been unused for the largest time is selected for eviction. Least-recently-used replacement requires recording of accesses to cache-lines and is therefore harder to implement.

If the processor issues a write request to a location in main memory, the cache checks whether it contains the contents of the desired location. In the case of a cache hit, the contents in the cache are updated. There are two different ways to subsequently update the location in main memory: write-through and write-back. If the cache uses the write-through protocol, the contents of main memory are updated simultaneously with the contents of the cache. If the cache uses the write-back protocol, the contents in main memory are only updated in case the location is evicted from the cache.

In the case of a cache miss, only the contents of main memory are updated, if the cache does not use the write-allocation protocol. Otherwise the contents are fetched from main memory and stored in the cache, evicting other cache-lines as necessary. Afterwards the contents in the cache are updated in the same way as for a write hit.

Cache Organization. One important aspect of caches is the mapping of memory location to cache-lines, i.e. where a given memory location can be placed in the cache. Three different types of cache organizations exist, namely direct-mapped, n-way set-associative and fully associative.

In a direct-mapped cache, each memory location has exactly one place to reside, usually the cache-line that is specified by the address of the memory location modulo the number of cache-lines. Caches of this type are easy to implement, since only one cache-line has to be checked in order to determine whether the contents of a given memory location are already in the cache. In addition, direct-mapped caches do not use one of the replacement strategies mentioned above, as the cache-line to be replaced is unique. However, the miss rates of caches using this organization are usually higher than the miss rates using any of the other organizations.

In a n-way set-associative cache, the cache is organized in sets, each set contains n entries. Memory locations are mapped to the individual sets by the address of the memory location modulo the number of sets, but can reside in any of the n entries in the set. Caches of this type are more expensive than direct-mapped caches, since n entries have to be checked in parallel in order to determine whether the contents of a given memory location are already in the cache. The parameter n is therefore usually quite small, i.e. $n = 2, 4, 8$. As a rule of thumb, the miss rate of a 2-way set-associative cache is the same as for an direct-mapped cache with doubled size [HP96].

In a fully associative cache, a location in memory can be placed in every entry of the cache, i.e. there are no restrictions in the mapping of memory locations to cache-lines. Caches of this type are very expensive, as all entries have to be checked in parallel in order to determine whether the contents of

a given memory location are already in the cache. Therefore fully-associative memories are usually quite small.

Caches can be used to store instructions, data, or a combination of both: An instruction cache holds only memory locations that contain program code, whereas a data cache holds only memory locations that contain data. A unified cache holds both types of memory locations. Current microprocessors usually use separate first-level instruction and data caches and a larger unified second-level cache, sometimes backed by an even larger external third-level cache.

Cache Misses. There are three different kinds of cache misses: compulsory, capacity, and conflict misses [HP96]. Compulsory misses are caused by accesses to memory locations that have never been accessed before, i.e. have never been in the cache. Capacity misses are caused by accesses to memory locations that have been in the cache before but have been evicted from the cache in the meantime due to the insufficient number of entries in the cache, i.e. the cache is too small to hold all the memory locations needed by the program. Conflict misses are caused by accesses to memory locations that have been in the cache before but have been evicted from the cache in the meantime due to the sharing of cache-lines, i.e. some of the intermediate accesses were to memory locations that mapped to the same cache-line.

Emulated Multithreading and Caches. Caches are essential for achieving high performance on current microprocessors. The interaction between emulated multithreading and caches is therefore important to the overall performance of emulated multithreading. Potential problems include the following:

- Emulated multithreading merges the program code with context switch code, which leads to an increase in code size. This may lead to increased cache miss rates for the instruction caches, as the instruction cache is usually already too small to hold the original code. The impact of this issue will depend on the application code as well as the conversion process.
- Emulated multithreading executes several threads that execute the same program code. If the threads operate in different areas of the program, this may lead to an increased number of conflict and/or capacity misses in the instruction cache. The impact of this issue will depend on the application code, the conversion process, as well as the number of threads.
- Emulated multithreading traverses the loop in the thread execution routine once for every context switch. The code of this loop should therefore always reside in the first-level instruction cache. As the main loop is executed fairly often, this can be expected.
- Emulated multithreading stores the contexts of the individual threads in memory. The converted program, especially the context switch code, will access these data structures fairly often. The data structures are therefore likely to reside in the first-level data cache, which may lead to an increase

in the number of conflict and/or capacity misses in the cache. However, the thread execution routine can be extended to prefetch the thread descriptors if the underlying architecture supports prefetching. Again, the impact of this issue will depend on the application code, the working set, the conversion process, the size of the data structures, as well as the number of threads.

Experimental results will be necessary to answer the question whether these or other problems exist, and to which extent they impact the performance of emulated multithreading. This question is discussed in Chapters 5 and 6, which cover the experimental results of emulated multithreading on a set of benchmarks on the Compaq XP1000 workstation and the Cray T3E massively parallel processor, respectively.

2.3.3 Branch Prediction

In order to obtain high performance on current microprocessors, it is necessary to avoid pipeline stalls, especially for modern multiple-issue and out-of-order processors. Control dependencies are a major source of pipeline stalls as branch instructions are frequently used: in typical integer code, one out of six instructions is a branch [Wal92][SJH89]. In addition, these dependencies can only be resolved late in the pipeline, which forces the flushing of all subsequent instructions in the pipeline if the instruction stream is discontinued. Branch prediction is used to resolve control dependencies early in the pipeline, thus avoiding or at least minimizing pipeline stalls, provided that the prediction is correct. Current microprocessors therefore invest considerable resources for branch prediction logic.

Recall that there are three types of control-flow instructions: conditional branches, unconditional branches and indirect branches. Conditional branches test a condition and perform a branch relative to the program counter if the tested condition is true. Unconditional branches perform a branch relative to the program counter without testing any condition. Indirect branches perform an unconditional branch, the target location is specified by a register operand. Indirect branches are mostly used for subroutine calls and returns. Note that conditional branches require the prediction of the branch direction (taken/not-taken) as well as the branch target, whereas the latter is sufficient for unconditional and indirect branches. The following paragraphs describe several branch prediction strategies with a special emphasis on strategies that are found in current microprocessors. More complete surveys of branch prediction strategies can be found in [HP96] [HCC89][MH86][LS84][Smi81b].

Branch Direction. Static prediction is the simplest way to predict the direction of a branch: The outcome of a branch is predicted independent of program behavior, i.e. the outcome of previous executions of the branch. The simplest static strategy is to predict all branches as taken, resulting in

a prediction accuracy of 41 % to 91 % for a subset of the SPEC92 programs [HP96]. Another variant predicts all forward branches as not-taken and all backward branches as taken. However, as usually more than 50 % of the branches are taken, this strategy is unlikely to achieve a prediction accuracy greater than 60 % [HP96].

As a prediction accuracy of 50 % corresponds to random prediction, i.e. static branch prediction is not very successful, at least for certain types of programs. However, the prediction accuracy for static approaches can be increased to 78 % - 95 % for the same subset of the SPEC92 programs by using profiling information from previous runs to guide the code generation process [HP96][FF92].

Dynamic prediction strategies predict the branch direction based on previous executions of a branch, i.e. the branch history. One such strategy uses a branch history table with m entries, where each entry contains a n-bit saturating counter to record the branch history. Given a specific branch, the direction of the branch is predicted as follows: The address of the branch to be predicted is used to select an entry in the branch history table, usually specified by the branch address modulo m. If the value of the corresponding counter is larger than 2^{n-1}, the branch is predicted as taken, otherwise it is predicted as not-taken. After the branch has been resolved, the branch history table is updated by incrementing the counter if the prediction was correct, decrementing otherwise. Different branches can map to the same entry in the branch history table, as long as the low-order address bits are identical. The accuracy of these branch predictors does not increase significantly beyond $n = 2$, hence $n = 2$ is commonly used. The prediction accuracy for a branch history table with 4096 entries and 2-bit saturating counters ranges between 82% and 100% for the SPEC89 programs [HP96].

A more accurate version of dynamic branch prediction based on the branch predictor described above are two-level or correlated branch predictors [PSR92][YP92]. A (m,n) correlated branch predictor combines 2^m branch history tables using n-bit saturating counters as described above with a m-bit history register that records the outcome of the last m branches. The contents of this history register are used to select a branch history table, afterwards the prediction proceeds as described above. This description covers only the basic properties of this type of branch predictor, several different versions are investigated in [YP93]. The accuracy of the two-level or correlated branch predictors ranges between 89 % and 100 % for a subset of the SPEC89 benchmarks [HP96].

Combining branch predictors [McF93] are an even more complicated branch prediction strategy that is used in recent microprocessors, e.g. the Alpha 21264. The basic idea behind combined predictors is to use three different branch predictors: a local, a global, as well as a choice branch predictor. The choice predictor is organized like a branch prediction table, but the result of the prediction is used to select between the local and global branch

predictors. The local and global branch predictors can be of any type, but usually two-level branch predictors are used with a single history register for the global predictor and a branch history table for the local predictor. The average accuracy of the combined branch predictors is 98.1 % for the SPEC89 programs, compared to 97.1 % for the two-level branch predictors [McF93].

Branch Target. The target of a branch is usually predicted with a branch target buffer [LS84]. The branch target buffer is a small cache that contains the address of a branch as well as the most recent target address in each entry. The target of a given branch is predicted as follows using a branch target buffer: If a branch is executed, the branch target buffer is looked up with the address of the branch instruction. If a corresponding entry is found in the branch target buffer and the branch is predicted as taken, the target address stored in the branch target buffer is used to predict the target of the branch. If no corresponding entry can be found, a new entry is allocated. After the branch has been resolved, the branch target buffer is updated with the actual branch target address. The accuracy of branch target buffers depends on the size of the buffer as well as the replacement strategy, prediction accuracies of 95.2 % for a buffer with 512 entries have been reported [LS84].

Return-address stacks are used to predict the target of indirect branches [KE91]. A return-address stack is a small stack that stores return addresses and works as follows: A subroutine call pushes the updated program counter, i.e. the address of the next instruction, onto the return-address stack. These return addresses are used by the subroutine returns, which pop the top-most return address from the stack and use this address to predict the target address. A special case are coroutine calls which perform both operations, i.e. pop the top-most entry to predict the call target and push the address of the next instruction onto the stack. As long as the return-address stack is larger than the maximum call-depth, the return-address stack will predict all return addresses correctly. Note that procedure calls themselves can be predicted using branch target buffers with reasonable accuracy.

Emulated Multithreading and Branch Prediction. Emulated multithreading uses a procedure call and return instruction pair to execute an instruction block of a given thread, i.e. significantly increases the number of executed subroutine calls and returns compared to the original program. The conditional and unconditional branches in the program are either unchanged or replaced by instruction sequences that contain a subroutine return instruction. In the case of floating-point conditional branches, these instruction sequences contain the original conditional branch as well, since there are no conditional moves for floating-point operands. Hence the number of executed subroutine return instruction increases even further. The number of executed unconditional branches will decrease in the same way as these branches are replaced by the instruction sequences mentioned above. The number of executed conditional branches increases due to the conditional branch in the main loop of the thread execution routine that is executed for each instruction

block. However, this branch is only taken if a thread has finished execution and is removed from the list of active threads, hence even simple dynamic branch predictors will predict the direction of this branch correctly.

Of greater concern are the subroutine return instructions at the end of each instruction block and the subroutine call in the main loop of the thread execution routine. Note that all subroutine returns at the end of an instruction block return to the same location, i.e. the instruction that follows the subroutine call in the main loop. Since the target address never changes, the target address of these subroutine returns can be predicted with reasonable accuracy using a branch target buffer. Using a return-address stack is even more effective: The subroutine return at the end of an instruction block will always be predicted correctly using a return-address stack, since the subroutine call in the main loop pushes the correct address on the stack. Note that code using emulated multithreading has a maximum call depth of one relative to the main loop, hence the size of the return-address stack is of less concern and can only be exceeded by calls to subroutines that do not use emulated multithreading.

The subroutine call in the main loop of the thread execution routine is problematic: The target of this call will likely be different upon every execution, unless one or more threads subsequently execute the same instruction block. This behavior renders a branch target buffer useless: Since the subroutine call is executed fairly often, it is likely that a corresponding entry in the branch target buffer will be created. However, the predicted target address will be wrong in most cases, each time causing a mispredict penalty.

Likewise, a return-address stack is useless, since the subroutine calls only push values onto the stack, i.e. a return-address stack is only useful for predicting subroutine returns. A possible alternative is to replace the subroutine call in the main loop as well as the subroutine returns at the end of each instruction block with a coroutine call. In this way, the return-address stack can be used to predict the target of the subroutine call in the main loop, the subroutine returns are predicted as described above. However, the prediction of the subroutine call will only be correct if there is a single thread that executes a linear sequence of instruction blocks. Otherwise the subroutine return instructions will push the wrong addresses on the return-address stack. This restriction conflicts with the purpose of emulated multithreading, i.e. to use several threads to hide the latency of accesses to remote memory locations.

On the other hand, the target of the subroutine call in the main loop of the thread execution routine is already stored in the thread descriptor, i.e. is well known in advance. On architectures that support branch prediction instructions, the subroutine call could be predicted by issuing a branch prediction hint prior to the call. Since branch prediction instructions must be issued a minimum distance from the branch to be effective, these instructions could be used to predict the target of the next but one thread, similar to the prefetching of thread descriptors.

2.3.4 Code Scheduling

Static and dynamic hazards are the cause of pipeline stalls that decrease the performance of pipelined microprocessors, especially in the case of multiple instruction issue. Code scheduling is used by the compiler to reduce the number of static data and structural hazards in the generated code, thereby increasing performance. The importance of code scheduling is illustrated by the fact that all modern compilers employ more or less aggressive forms of code scheduling. Note that code scheduling is usually NP-hard [GJ79], hence most algorithms are based on heuristics. Popular approaches are global instruction scheduling [BR91] and selective scheduling [ME97]. A good introduction to various code scheduling techniques can be found in [Muc97].

Emulated multithreading affects code scheduling, since the conversion process merges tailored context switch code with the application program. The conversion process inserts save and restore instructions into the original program, which is likely to render the previous code scheduling useless, hence the converted code should be scheduled again after the conversion process. If the assembler supports code scheduling optimizations, this rescheduling can be performed by the assembler, otherwise a code scheduling pass has to be added to the conversion program.

2.3.5 Out-of-order Execution

Out-of-order execution is a technique used to extract more instruction-level parallelism from the instruction stream: A processor that supports out-of-order execution issues instructions out of sequential program order to the execution units, provided that the instructions are independent. Some processors combine out-of-order issue with speculation, i.e. issue instructions out-of-order even if the independence has not been established. Note that out-of-order execution implies out-of-order completion, i.e. instructions do not finish in sequential order.

The goal of the instruction fetch stage is to find enough independent instructions that can be issued in parallel. The limiting factor is the size of the instruction window, i.e. the number of instructions that are inspected for dependencies. On the one hand, a larger instruction window will increase the potential to find independent instructions. On the other hand, a larger instruction window requires accurate branch prediction in order to fill the instruction window correctly due to the frequent use of branches.

The issue stage forwards the instructions to the execution stage, provided that enough execution units are available. The scheduling phase assigns the issued instructions to designated execution units at designated times. There are two different forms of scheduling:

- Control-flow scheduling [Tho70] does not issue instructions until all data and resource dependencies have been resolved and is usually implemented with a central resource called the scoreboard.

- Data-flow scheduling [Tom67] issues instructions immediately to the execution units, where the instructions are stored in buffers until all resources are available. Data-flow scheduling is implemented using distributed resources and is usually combined with register renaming [Sim00].

Emulated multithreading benefits from out-of-order execution, since there are no data dependencies between individual threads: All registers used in an instruction block are initialized upon first use either via a restore operation or via a normal instruction. A processor that uses register renaming will be able to detect the absence of data dependencies. Apart from the data dependencies within the instruction blocks, the only data dependencies between the instruction blocks and the main loop of the thread execution routine exist via the ThreadPC register, since that register is updated at the end of the instruction block and is used inside the main loop.

Control dependencies exist between instruction blocks and the main loop via the subroutine return and between other instruction blocks via the main loop. As long as the processor fills the instruction window across predicted branches and the branches are predicted correctly, these control dependencies present no problems. As mentioned above, the subroutine returns at the end of the instruction blocks will be predicted correctly using a return-address stack, but the subroutine call to the next instruction block will always be mispredicted in the absence of specific branch prediction instructions. Unless the subroutine calls can be predicted correctly, out-of-order execution will only cover the current instruction block and the loop in the thread execution routine, it will not cover the next instruction block. Therefore accurate branch prediction is important to the performance of emulated multithreading.

2.4 Architecture Support

The previous section has described the interaction between emulated multithreading and the underlying architecture. During this discussion, a set of architectural characteristics has emerged that improve the performance of emulated multithreading:

- The number and size of the integer and floating-point registers dominates the size of the thread descriptor. Larger thread descriptors could increase problems with the data caches, thereby affecting performance. The number of the integer and floating-point registers should therefore not be too large.
- The number of special registers increases the size of the thread descriptor as well as the performance of the generated code, e.g. if exception barriers are required to save and restore these registers. The number of these registers should therefore be quite small.
- Prefetch instructions can be used to prefetch thread descriptors in the main loop of the thread execution routine, potentially increasing performance.

An architecture that supports such instructions is therefore beneficial to emulated multithreading.

- Branch prediction hints are essential to predict the target of the subroutine call in the main loop of the thread execution routine correctly. Since this subroutine call will be executed fairly often, this can have a significant impact on performance. An architecture that supports such instructions is therefore beneficial to emulated multithreading.
- The emulation library contains synchronization routines that require some form of atomic read-modify-write instructions, e.g. load-locked and store-conditional, in order to implement locks and barriers.
- Out-of-order execution potentially improves the performance of programs using emulated multithreading, especially in combination with branch prediction hints. Implementations that support out-of-order execution are therefore beneficial to emulated multithreading, hence such implementations should exist for the underlying architecture.
- Emulated multithreading was designed to hide the latency of remote memory access in massively parallel processors. Such machines should therefore exist for a given architecture.

The characteristics of six currently popular commercial processor architectures are summarized in Table 2.1. The following paragraphs discuss the suitability of each of these architectures in detail before choosing the architecture that is used for the first implementation of emulated multithreading.

The Power architecture is a 64 bit architecture that defines 32 integer as well as 32 floating-point registers, each 64 bits wide. There are no dedicated zero source and sink registers: Register r0 provides a zero operand only if used as a base register during addressing. There are five special registers: the condition, link and count registers, the integer exception register as well as the floating-point status and control register. Instructions have a fixed size of 32 bits, while byte, word, longword, and quadword integers as well as single- and double-precision floating-point formats are supported. The architecture defines prefetch and synchronization instructions and supports static branch prediction for conditional branches. All implementations of the Power architecture, e.g. Power1, Power2, Power3, have supported out-of-order execution right from the start. The IBM SP2 is a massively-parallel processor based on the Power architecture and supports thousands of processors. Information about the Power and PowerPC architectures can be found in [MSSW94].

The Sparc V9 architecture is a 64 bit architecture that defines 32 integer as well as 32 floating-point registers, each 64 bits wide. Note that the Sparc architecture uses overlapping register windows. Register r0 is a zero source and sink register, all other registers are general-purpose. There are four special registers: the multiply/divide register, the condition code register, the floating-point state register, and the address space identifier. Instructions have a fixed size of 32 bits, while byte, word, longword, and quadword integers as well as single- and double-precision floating-point formats are sup-

Table 2.1. Comparison of RISC Architectures

	Power	Sparc V9	IA64
Int Registers (Number)	32	32	128
Int Registers (Size)	64 bit	64 bit	64 bit
FP Registers (Number)	32	32	128
FP Registers (Size)	64 bit	64 bit	82 bit
Special Registers	5	4	205
Instruction Format	32 bit	32 bit	128 bit
Data Formats	8,16,32,64 bit	8,16,32,64 bit	8,16,32,64 bit
Prefetch Instructions	√	√	√
Synchronizations	√	√	√
Branch Hints	(√)	(√)	√
Out-of-Order Impl.	√	√	–
MPPs ≥ 64 PEs	√	(√)	–

	HP-PA 2.0	MIPS IV	Alpha
Int Registers (Number)	32	32	32
Int Registers (Size)	64 bit	64 bit	64 bit
FP Registers (Number)	32	32	32
FP Registers (Size)	64 bit	64 bit	64 bit
Special Registers	8	3	1
Instruction Format	32 bit	32 bit	32 bit
Data Formats	8,16,32,64 bit	8,16,32,64 bit	8,16,32,64 bit
Prefetch Instructions	√	√	√
Synchronizations	√	√	√
Branch Hints	–	(√)	(√)
Out-of-Order Impl.	√	√	√
MPPs ≥ 64 PEs	(√)	√	√

ported. The architecture defines prefetch and synchronization instructions and supports static branch prediction for conditional branches. The UltraSparc processors from Sun do not support out-of-order execution, but the Fujitsu Sparc64-III is an out-of-order implementation of the Sparc architecture. The Sun UltraEnterprise 10 000 is the largest multi-processor based on the Sparc architecture and supports up to 64 processors. Moreover, the successor machine based on the UltraSparc-III will support hundreds of processors. Information about the SPARC architecture can be found in [Cat91][WG93].

The Intel IA64 architecture is a 64 bit VLIW architecture that defines 128 integer as well as 128 floating-point registers, each 64 and 82 bits wide, respectively. There is a large number of special registers: 64 single-bit predicate registers, eight branch registers, 128 application registers, the register stack configuration register, the floating-point status register, the loop and epilog count registers, as well as the user mask. Instructions have a fixed size of 128 bits and contain up to three instructions, while byte, word, longword, and quadword integers as well as single- and double-precision floating-point formats are supported. The architecture defines prefetch and synchronization instructions and supports special branch prediction instructions, which can be used to manipulate the return-address stack. The first implementation

of the IA64 architecture, the Itanium, does not support out-of-order execution. As the architecture is rather new, there are only workstations and small servers, but no massively-parallel processors available. Information about the IA64 architecture can be found in [Ita01].

The HP-PA 2.0 architecture is a 64 bit architecture that defines 32 integer as well as 32 floating-point registers, each 64 bits wide. Register r0 is a zero source and sink register, all other registers are general-purpose. There are eight special registers: five space registers, the shift amount register, the branch nomination register, as well as the floating-point status register. Instructions have a fixed size of 32 bits, while byte, word, longword, and quadword integers as well as single- and double-precision floating-point formats are supported. The architecture defines prefetch and synchronization instructions. The PA-8x00 line of processors is based on the HP-PA 2.0 architecture and supports out-of-order execution. The HP SuperDome is the largest multiprocessor based on the HP-PA architecture and supports up to 64 processors. Information about the HP-PA architecture can be found in [Kan95].

The MIPS-IV architecture is a 64 bit architecture that defines 32 integer as well as 32 floating-point registers, each 64 bits wide. Register r0 is a zero source and sink register, all other registers are general-purpose. There are three special registers: the high and low registers as well as the floating-point status register. Instructions have a fixed size of 32 bits, while byte, word, longword, and quadword integers as well as single- and double-precision floating-point formats are supported. The architecture defines prefetch and synchronization instructions and supports static branch prediction. The R10 000, R12 000 and R14 000 processors are out-of-order implementations of the MIPS architecture, the SGI Origin 2000 and 3000 are massively parallel systems that support hundreds of processors. Information about the MIPS architecture can be found in [KH92].

The Alpha architecture is a 64 bit architecture that defines 32 integer as well as 32 floating-point registers, each 64 bits wide. Registers r0 and f0 are zero source and sink registers, all other registers are general-purpose. There is only one special register, i.e. the floating-point control register. Instructions have a fixed size of 32 bits, while byte, word, longword, and quadword integers as well as single- and double-precision floating-point formats are supported. The architecture defines prefetch and synchronization instructions and supports static branch prediction. The 21264 processor was the first out-of-order implementation of the Alpha architecture, the Cray T3D, T3E and the Compaq AlphaServer SC are massively parallel systems that support thousands of processors. Information about the Alpha architecture can be found in [Com98].

The Alpha architecture was chosen for the first implementation of emulated multithreading since it is best suited for this technique: The number and size of the integer and floating-point registers is comparable to other RISC architectures, but there is only one special register which is only infrequently

used. The architecture supports prefetch instructions, synchronization instructions, as well as static branch prediction. Implementations of the Alpha architecture make extensive use of multiple issue and out-of-order execution, e.g. Alpha 21264, and have always been the world's fastest microprocessor according to the SPEC benchmarks. In addition, several massively parallel processors based on the Alpha architecture exist and are in widespread use, e.g. the Cray T3E and the Compaq AlphaServer SC. The only major drawback of using this architecture is the absence of explicit branch prediction instructions similar to the IA64 architecture. The next chapter discusses the current implementation of emulated multithreading on the Alpha architecture.

3. Implementation

The last chapter has described the basic concept of emulated multithreading. This chapter covers the current implementation of emulated multithreading, special emphasis is placed on the tools and algorithms used in the code conversion process. The chapter is organized as follows: Section 3.1 introduces the capabilities of the implementation, while Section 3.2 describes the design flow and the tasks of the individual tools. These tools are described in detail in Sections 3.3, 3.4 and 3.5. Section 3.6 covers register partitioning, an enhanced optimization technique. Section 3.7 covers platform-specific implementation details, while Section 3.8 discusses the benefits of integrating the individual tools into an existing compiler.

3.1 Introduction

As described in the previous chapter, code conversion is the major element of emulated multithreading. The code conversion process is performed after code generation, i.e. the conversion process operates on assembler code or an equivalent low-level representation. Tools that operate on these kind of representations are called postpass-optimizers [Kae00]. There are two basic ways to implement such postpass-optimizers: They can be integrated into the compiler or they can be implemented as stand-alone tools that operate on assembler source files. The next paragraphs discuss the benefits and drawbacks of both approaches.

Integrating the conversion process into the compiler allows the conversion process to utilize the existing compiler infrastructure, e.g. register allocation, instruction scheduling, optimization passes, as well as the corresponding data structures. In addition, it is possible to instruct other compilation phases to generate code that is better suited for the conversion. This phase-combining approach was shown to be effective in other areas, e.g. integrating scheduling and register allocation [BEH91]. A major drawback of the integrated approach is that the source code of the compiler is usually needed for a successful integration. However, at least for commercial products the source code is not available. The notable exception is the Gnu Compiler Collection [LO96], but this compiler suite is not available on all of the platforms used during the

evaluation of emulated multithreading. An alternative is the Stanford University Intermediate Format (SUIF) compiler system [WFW+94][Lam99]. The integration of the conversion process into this highly modular compiler system is discussed in Section 3.8.

Implementing the conversion process as a stand-alone tool allows the use of different compilers: As long as the compilers use the same assembler syntax, the same conversion tool can be used, since the compilers generate code for the specified assembler. Even if the compilers use different assemblers on the same platform, at most those parts of the conversion tool that parse the assembler code needs to be changed.

Another benefit is the portability of separate tools: it is not necessary to port a whole compiler to support a new platform, only the conversion tool itself has to be ported. Even a separate tool can reuse the existing programming environment, e.g. leave the instruction scheduling phase to the native assembler. A major drawback of the stand-alone approach is the need to reimplement at least parts of the compiler structure, e.g. lexer, parser, register allocation. The separate conversion tool is therefore more complex to implement, but is not constrained by the structure of the existing compiler.

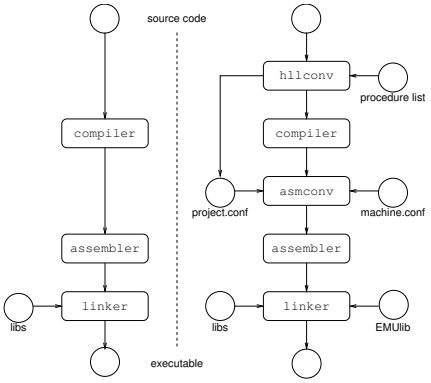
The code conversion process was implemented as a stand-alone tool for several reasons: First, the SUIF compiler system, especially the machineSUIF backend was undergoing major revisions, making it hard to decide whether such an integration would be feasible. For example, the optimization programming interface was totally undocumented at the time. Second, the implementation of emulated multithreading should use the native compiler on each platform in order to ensure that the results are not biased by using another compiler that might generate suboptimal code.

The code conversion process is implemented in a stand-alone tool called asmconv. This tool operates on the assembler source generated by the compiler and performs the code conversion process. The assembler converter is augmented by another converter called hllconv that operates on the high-level language. This converter is used to generate configuration files for the assembler converter. In addition, the high-level language converter performs the required procedure duplication and call substitutions on the high-level language source code, if register partitioning as described in Section 2.1.3 is used. The next section discusses the design flow as well as the capabilities of the converters in detail.

3.2 Design Flow

The code conversion process is based on two stand-alone tools, i.e. the asmconv and hllconv converters, as well as the existing programming environment, i.e. compiler, assembler, linker. The design flow and interaction between these tools is illustrated in Figure 3.1. The left part of the figure depicts the

Fig. 3.1. Design Flow



design flow for modules that contain no procedures using emulated multithreading. The right part of the figure shows the design flow for modules that contain at least one such procedure.

In the standard design flow, the source code of the module in a language such as C, C++, or Fortran is passed to the corresponding compiler. The compiler generates a source file in assembly language for the module. This source file is subsequently passed to the assembler, which generates the corresponding object file. After all object files have been generated, the object files as well as the referenced libraries are processed by the linker, which produces the corresponding executable.

The standard design flow described above was slightly modified to incorporate the conversion process: The high-level language converter is executed before the compiler, while the assembler converter is executed after the compiler, but before the assembler. The converted code references routines from the emulation library, hence this library is passed to the linker in addition to the object files and the other libraries. A detailed description of the library can be found in Section 3.4.

The high-level language converter parses the source code, searching for procedure definitions, declarations or calls to procedures from list of procedures to be converted. Based on this information, the high-level language converter generates a project-specific configuration file for the assembler converter and performs procedure duplication and call substitution in the case of register partitioning. A detailed description of the converter is given in Section 3.3.

The assembler converter operates on the assembler source generated by the compiler. Based on the information in the platform-specific configuration file (machine.conf) as well as the module-specific configuration file generated by hllconv, the assembler converter performs the code conversion process on selected procedures as described in Section 2.2.4. The converter generates

a new assembler source file that contains the converted code and which is transformed into an object file by the assembler. A detailed description of the converter is given in Section 3.5.

3.3 High-Level Language Converter

The high-level language converter hllconv is responsible for generating a module-specific configuration file for the assembler converter as well as procedure duplication and call substitution in the case of register partitioning. Section 3.3.1 describes the structure and contents of this configuration file, while Sections 3.3.2 and 3.3.3 cover the conversion tasks of the high-level language converter as well as the implementation, respectively.

3.3.1 Configuration File

The module-specific configuration file contains information about every procedure to be converted as well as every procedure called by such an procedure. Since the assembler converter obtains information about system and other platform-specific calls from the platform-specific configuration file, the high-level language converter can omit these calls. For each procedure, the configuration file contains an entry that consists of a name and type as well as several register fields.

The name field consists of the name keyword followed by a string enclosed in quotes that contains the name of the procedure. The type field consists of the type keyword followed by a space-separated list of keywords: The intern keyword marks the procedure as internal, the code conversion process will be applied to all internal procedures. External procedures, e.g. system calls, are marked by the extern keyword. Either the intern or extern keywords have to be specified. Calls to procedures marked with the switch keyword force the call instruction to end an instruction block, i.e. initiate a context switch. The switch keyword is mandatory for internal procedures and optional for external procedures. The fixarg keyword is used for procedures with a fixed number of arguments, variable-argument procedures use the vararg keyword.

A register field can be of several types: The rreq register field consists of the rreq keyword followed by a string enclosed in quotes. The string contains a colon-separated list of register names, the designated registers are expected by the call, but are not the argument registers, e.g. frame and stack pointers. The ropt register field consists of the ropt keyword followed by a string enclosed in quotes. The string contains a colon-separated list of register names, the designated registers are expected by the call, but are not argument registers. The rarg register field consists of the rarg keyword followed by a string enclosed in quotes. The string contains a colon-separated list of register names, the designated registers are the argument registers of

the call. The string must be empty for procedures that use a variable number of arguments. The rret register field consists of the rret keyword followed by a string enclosed in quotes. The string contains a colon-separated list of register names, the designated registers are the return registers of the call.

3.3.2 Conversion Tasks

The primary task of the high-level language converter is the creation of the configuration file described above. This configuration file has an entry for each procedure, whose name is in the list of user-supplied procedure names. These procedures are called internal procedures because the corresponding assembler code will be converted by the assembler converter. In addition, the configuration file has an entry for every procedure that is called by an internal procedure.

The necessary information can be gathered by inspecting the declaration and body of all internal procedures and the declaration of all procedures called by internal procedures. The procedure declaration contains information about the number and type of arguments and return values. Together with the platform-specific calling conventions, this information can be translated to the four individual register fields described above. Hence the platform-specific calling conventions must be incorporated into the high-level language converter. Inspection of the procedure body reveals all calls to external or internal procedures. The information for the external procedures can be gathered in the same way as for internal procedures, i.e. by inspecting the corresponding procedure declaration.

If register partitioning is used, the high-level language converter has to perform the necessary code duplication and call substitutions: For every internal procedure, the corresponding procedure declaration and body have to be duplicated p times, where p is the number of register partitions. The p additional procedures are distinguished from the original procedure by adding a prefix as well as an postfix to the procedure name: The prefix is a fixed string, while the postfix contains the number of the partition. Note that the original procedure must be preserved, since it may be called by one of the external procedures. Inside the duplicated procedure bodies, calls to internal procedures must be substituted with calls to the partition-specific copies of these procedures.

3.3.3 Implementation

The transformations described above are easy to perform on the call graph:

Definition 3.3.1. *The* call graph *of a program P containing the procedures $p_1, \ldots, p_n$ is the graph $G = (N, S, E, r)$, where $N = \{p_1, \ldots, p_n\}$ is the set of nodes, S is the set of call sites denoted by byte offsets, r is the node that contains the program entry point and $E \subset N \times N \times S$ is a set of labeled edges.*

Each $e = (p_i, s, p_j) \in E$ represents a call of procedure p_j at the call site s in procedure p_i.

Constructing such a graph is straight-forward except for two problems: separate compilation and procedure pointers. If the program P consists of multiple modules that are compiled separately, the call graph must be constructed incrementally or all modules have to be processed simultaneously. The second problem is more serious, since the existence of procedure pointers makes the construction of a call graph PSPACE-hard [Wei80], hence usually a conservative approximation of the call graph is used. At the moment, procedure pointers are not allowed.

Given the call graph G, the transformations described above can be performed as follows: Let I be the set of internal procedures given by

$$I = \{p \in N | \text{ the name of procedure } p \text{ is in the user-supplied list}\}$$

Based on the set I, the set X of external procedures can be described as

$$X = \{p_j \in N \,|\, \exists (p_i, s, p_j,) \in E : p_i \in I \wedge p_j \notin I\}$$

Note that system calls are omitted by default, since only the set N of procedures in the program itself are inspected. After the sets I and X have been determined, it is sufficient to translate the procedure declaration for each procedure in both sets into an entry in the configuration file. This translation is straight-forward if the calling conventions are known.

If register partitioning is used, procedure duplication and call substitution have to be performed. These operations can be implemented by transforming the call graph as follows: Given the call graph $G = (N, S, E, r)$ and the set I of internal procedures, let p be the number of partitions. Duplicating all internal procedures yields the call graph $G' = (N', S', E', r')$, where

$$\begin{aligned}
G' &= G \cup \{p_i^k \,|\, p_i \in I,\, 1 \le k \le p\} \\
S' &= S \cup \{s_j^k \,|\, \exists (p_i, s_j, p_j) \in E \,:\, p_i \in I,\, 1 \le k \le p\} \\
E' &= E \cup \{(p_i^k, s^k, p_j) \,|\, \exists (p_i, s, p_j) \in E\,;\, p_i, p_j \in I\,, 1 \le k \le p\} \\
&\quad \cup \{(p_i^k, s^k, p_j^k) \,|\, \exists (p_i, s, p_j) \in I \,:\, p_i, p_j \in I\,, 1 \le k \le p\} \\
r' &= r
\end{aligned}$$

The call graph transformations described above provide the basis for the implementation of the high-level language converter. The converter therefore constructs the call graph of a program, performs the transformation described above on the call graph, and creates the configuration file afterwards. The high-level language converter inspects all modules of the program simultaneously to address the issue of separate compilation.

There are several call graph constructors available, a quantitative survey can be found in [MNL96]. Instead of implementing a call graph extractor in

the high-level language converter, the converter uses the output from one of these call graph extractors, i.e. cflow. This requires only a lexer and parser to translate the output of the extractor into an internal representation. The lexer and parser stage can be implemented using automatic code generators, e.g. lex [LMB92] and yacc [LMB92], respectively.

The translation of procedure declarations into types and register fields can be handled as follows: If the procedure is internal, the intern keyword is added to the type field, otherwise the extern keyword is added. If the procedure expects a variable number of arguments, the vararg keyword is added to the type field, otherwise the fixarg keyword is added. The required and optional register fields are determined by the calling convention.

The argument registers are determined as follows: If the procedure expects a variable number of arguments, the corresponding field is left empty. Otherwise the argument list in the procedure declaration is processed from left to right in order to determine the type, i.e. integer or floating-point, of each argument. Integer and pointer arguments are usually passed in the integer argument registers, while the floating-point arguments are usually passed in the floating-point argument registers. If the number of arguments exceeds the number of argument registers, the remaining arguments are usually passed via the stack.

After the type of all arguments has been determined, the individual arguments are mapped to the corresponding argument registers according to the calling conventions. For example, the Tru64 operating system assigns the ith argument to the ith integer or floating-point argument register, depending on the type of the argument. Note that this assignment uses either the ith integer or the ith floating-point register, but never both at the same time, hence does not utilize all available argument registers in all situations.

The return register field is determined in the same way as the argument register fields by inspecting the return type of the procedure as given by the procedure declaration and assigning the corresponding return value register according to the calling convention.

3.4 Emulation Library

The emulation library contains routines that are called by the transformed code and contains three different groups of routines: thread initialization, thread execution, and communication routines. As the implementation of almost all routines in the library is platform-specific, the following sections describe only the functionality as well as platform-independent implementation issues. The platform-specific details can be found in Section 3.7. Section 3.4.1 covers the thread initialization routines, while Section 3.4.2 describes the thread execution and local synchronization routines. The remote communication and synchronization routines are described in Section 3.4.3.

3.4.1 Thread Initialization Routines

The thread initialization routines initialize the required data structures mentioned in Section 2.2.2, e.g. the thread control block and the thread descriptors. Apart from these data structures, two additional data structures are used during initialization: The thread arguments structure is used to store the arguments for the thread and contains two arrays for the integer and floating-point arguments as well as the number of integer and floating-point arguments. The thread attribute structure is used to pass thread attributes, i.e. stack size and number of threads, to the thread creation routine.

The EMUthread_args_init() routine allocates and initializes a thread argument structure, while the EMUthread_args_destroy() routine frees the memory associated with a thread argument structure. The EMUthread_args_set() routine copies the individual arguments into the given thread argument structure and expects the corresponding integer and floating-point arguments as input. These arguments are copied to the integer and floating-point argument arrays of the thread arguments structure according to the calling convention. Note that the routine currently does not support the passing of arguments via the stack, i.e. all arguments for the thread startup procedure have to be passed via registers.

The EMUthread_attr_init() routine allocates and initializes a thread attribute structure, while the EMUthread_attr_destroy() routine frees the memory associated with a thread attribute structure. The EMUthread_attr_setnumthreads() routine expects the number of threads as an argument and initializes the corresponding field of the specified thread attribute structure, while the EMUthread_attr_getnumthreads() routine returns the number of threads as stored in the specified thread attribute structure. The EMUthread_attr_setstacksize() routine expects the stacksize for the individual threads as an argument and initializes the corresponding field of the specified thread attribute structure, while the EMUthread_attr_getstacksize() routine returns the stacksize as stored in the specified thread attribute structure.

The EMUthread_create() routine is used to create the thread control block as well as the thread descriptors for the individual threads. The routine expects a thread control block, thread argument, and attribute structures as well as a pointer to the thread startup procedure as arguments. The routine initializes the thread control block, allocates and initializes the thread descriptors for each thread. The number of threads as well as the stacksize are taken from the thread attribute structure. In order to reduce the number of memory allocations, the thread descriptors and thread stacks are allocated in two subsequent memory allocations.

Each thread descriptor is assigned a unique number in the range from 0 to $t-1$, where t is the number of threads. The integer and floating-point register fields are initialized by copying the arguments from the thread argument structure to the corresponding argument registers. In addition, the stack and

frame pointers are set to the threads stack and the return address register is initialized to the fourth step in the main loop of the thread execution routine. The program counter is set to the value of the procedure pointer.

The thread descriptors are organized in a circular, doubly-linked list, the head and tail pointers of the thread control block are initialized accordingly. After all thread descriptors have been initialized, the EMUloop() thread execution routine is used to start the execution of the threads.

3.4.2 Thread Execution Routines

The second group of routines in the emulation library is used during thread execution and contains the following four routines:

The EMUloop() routine is the thread execution routine that has already been described in Section 2.2.3. The routine is written in machine language for performance reasons and therefore platform-specific. The EMUthread_self() routine expects no arguments and returns the unique number that identifies the current thread. The EMUthread_switch() routine expects no arguments and returns immediately without performing any operations. The platform-specific configuration file instructs the assembler converter to force a context switch after each call to this routine.

The EMUthread_cswap() routine performs a conditional swap and expects three arguments, i.e. the address of a memory location as well as the condition and value arguments. If the contents of the specified memory location are equal to the condition value, an atomic read-modify-write sequence is used to store the value at the specified memory location. The routine returns the old contents of the memory location in either case. The EMUthread_cswap() routine is useful for implementing inter-thread synchronization locks.

The EMUthread_barrier() routine expects no arguments and performs a barrier synchronization among all threads. The barrier is not restricted to a single processor, but covers all threads on all processors. Upon entry of the barrier, a thread is removed from the active list and stored in the inactive list instead. After the last local thread has entered the barrier, a system-wide barrier is performed to synchronize all processors in the system. Afterwards the head and tail pointers of the active and inactive lists are simply exchanged, thereby effectively moving all local threads from the inactive to the active list. Recall that each processor maintains its own thread control block with local active and inactive lists.

3.4.3 Communication Routines

The third and last group of routines in the emulation library covers inter-processor communication and synchronization routines. These routines implement a split-transaction protocol, i.e. the initialization and completion of requests is separated in order to support latency tolerance. The separation is based on the concept of E-registers, i.e. external registers, that are

used to perform the actual data transfer. These E-registers are either implemented in hardware, e.g. Cray T3E, or in software, e.g. Compaq XP1000. The group consists of the EMUereg_get(), EMUereg_put(), EMUereg_cswap(), EMUereg_mswap(), EMUereg_finc(), EMUereg_fadd(), EMUereg_pending(), and EMUereg_state() routines.

The EMUereg_get() routine is available for several different data types and expects three arguments: the number of an E-register, the address of the memory location at the remote processor, as well as the number of the remote processor. The routine initializes a remote read operation to the specified memory location at the remote processor, storing the result in the specified E-register. The routine returns after initializing the read request, i.e. usually before the actual data transfer has been finished. The only difference between the versions of the EMUereg_get() routine is the amount of transferred data, which is equal to the size of the data type.

The EMUereg_load() routine is available for several data types and expects the number of an E-register as argument. The routine returns the content of the specified E-register converted to the corresponding data type. If the corresponding data transfer has not been finished, the routine waits until the data transfer is completed before returning the content of the E-register. This routine is used in conjunction with the other routines to complete requests that have been issued from one of the other routines.

The EMUereg_put() routine is available for several data types and expects four arguments: the number of an E-register, the address of the memory location at the remote processor, the value to store, as well as the number of the remote processor. The routine stores the specified value at the memory location of the remote processor, using the specified E-register to perform the data transfer. The routine returns after the remote write request has been initialized, i.e. usually before the actual data transfer takes place. Note that it is not necessary to wait for the completion of the remote write request, since such a request returns no result. The only difference between the versions of the EMUereg_put() routine is the amount of transferred data, which is equal to the size of the data type.

The EMUereg_cswap() routine performs a conditional swap operation similar to the EMUthread_cswap() routine described above, but operates on remote memory instead. The routine expects five arguments: the number of an E-register, the address of a memory location, the condition, value, as well as the number of the remote processor. If the original contents of the memory location at the remote processor are equal to the specified condition, the specified value is stored to this location The original contents of the remote memory location is returned in either case and is stored in the specified E-register. The routine returns after the conditional swap operation has been initialized, i.e. usually before the actual data transfer has been finished.

The EMUereg_mswap() routine performs a remote swap operation and expects four arguments: the number of an E-register, the address of a memory

location, the value, as well as the number of the remote processor. The routine stores the specified value to the remote memory location, simultaneously returning the original contents of the memory location in the specified E-register. The routine returns after the swap operation has been initialized, i.e. usually before the actual data transfer has been finished.

The EMUereg_finc() routine performs a remote fetch-and-increment operation and expects three arguments: the number of an E-register, the address of a memory location, as well as the number of the remote processor. The routine increments the contents of the remote memory location and returns the original contents of the location in the specified E-register. The routine returns after the fetch-and-increment operation has been initialized, i.e. usually before the actual data transfer has been finished.

The EMUereg_fadd() routine performs a remote fetch-and-add operation and expects four arguments: the number of an E-register, the address of a memory location, the addend, as well as the number of the remote processor. The routine adds the addend to the contents of the remote memory location and returns the original contents of the location in the specified E-register. The routine returns after the fetch-and-add operation has been initialized, i.e. usually before the actual data transfer has been finished.

The EMUereg_pending() and EMUereg_state() routines are used to obtain the E-register state: The former routine expects no arguments and returns a value that indicates whether any remote memory accesses are outstanding. The latter routine returns the state of the specified E-register. This is useful for determining whether any remote memory operations using this E-register are still outstanding.

3.5 Assembler converter

The assembler converter performs the actual code conversion as described in Section 2.2.4. The assembler converter operates on the assembler source generated by the compiler and uses several configuration files and command-line arguments to steer the conversion process. The structure of these configuration files as well as the various configuration options are described in Section 3.5.1.

The assembler converter uses two passes to parse the assembler source, although the first pass is optional, i.e. is only performed on some platforms. The task of the first pass is therefore described in Section 3.7 along with other platform-specific issues. During the second pass, internal procedures are detected and the corresponding instructions are translated into a sequence of internal data structures. The lexer and parsers used to process the assembler source are described in Section 3.5.2.

The sequence of instructions is subsequently grouped into basic blocks: A basic block is a maximal sequence of sequential instructions, i.e. is bounded by branch instructions and labels. In addition the maximum size of the basic

blocks can be limited via command-line options, thus forcing a new basic block after the current basic block exceeds the maximum number of instructions. The basic block creation process is described in Section 3.5.3.

Basic blocks can be grouped into larger super blocks if the corresponding optimization is enabled. A super block is a set of basic blocks that has a single point of entry, but can have multiple exit points. The super block creation is described in Section 3.5.4.

After creating the basic and super blocks, external calls are detected and the corresponding call prolog/epilog sequences are determined. This process includes the merging of basic and super blocks, if these sequences cross basic or super block boundaries. The handling of external calls is handled in Section 3.5.5.

After the final shape of the basic and super blocks has been determined, several data-flow analyses are performed, the registers are allocated. These data-flow analyses are covered in Section 3.5.6. The allocation of registers is described in Section 3.5.7.

After register allocation, the actual code conversion process is performed by updating the instruction sequence, modifying the individual instructions and writing the converted procedure to the output file. These steps are covered in Section 3.5.8.

The assembler converter produces detailed statistics about original and modified instructions in a basic block, super block, procedure, or module. The individual statistics are covered in Section 3.5.9.

3.5.1 Configuration

The assembler converter is configured via command-line arguments and several configuration files: a platform-specific configuration file and a module-specific configuration file. The individual command-line arguments, the structure and contents of the platform-specific configuration file are covered in the following section. The structure and contents of the module-specific configuration file has already been described in Section 3.3.1.

Command-Line Arguments. The assembler converter supports the following command-line arguments:

- The -f ⟨name⟩ argument is mandatory and specifies the name and location of the platform-specific configuration file.
- The -l ⟨name⟩ argument is mandatory and specifies the name and location of the module-specific configuration file.
- The -o ⟨name⟩ argument is optional and specifies the name and location of the output file, i.e. the converted assembler code is written to this file. If this option is omitted, the converter uses the standard output instead.
- The -g ⟨num⟩ argument is optional and specifies the maximum grainsize in instructions. If the grainsize is larger than zero, the size of basic blocks

is forced to be smaller than the specified number of instructions. Otherwise the size of basic blocks is only limited by branch targets and branch instructions.

- The -p ⟨num⟩ argument is optional and specifies the number of register partitions to use. The assembler converter supports between one and four partitions on the currently supported platforms. If this option is omitted, the individual threads share the whole register set.
- The -O ⟨type⟩ argument is optional and enables the specified optimizations. It is possible to specify multiple such arguments in order to enable several optimizations simultaneously. All optimizations are disabled by default, using one of the argument with a no prefix can be used to disable the specified optimization. The following optimization types are supported:
  - The ropt keyword enables or disables the optimization of register stores during the actual code conversion.
  - The rreg keyword enables or disables the random selection of registers during register allocation.
  - The sblk keyword enables or disables the super block optimization.
  - The trap keyword is platform-specific and enables or disables the optimization of traps and exception barriers during the code conversion process.
  - The fpcr keyword is platform-specific and enables or disables the optimization of save and restore operations to the floating-point control register.
  - The arch keyword is platform-specific and enables implementation-specific optimizations as described in Section 3.5.1.
  - The emufix keyword enables or disables the use of fixed registers for the ThreadPC and ReturnPC registers which is the default. Disabling this option causes these registers to be allocated like normal registers, which can be useful for large basic or super blocks with a high register pressure, as these registers are usually used only at the end of such a block.
- The -d ⟨num⟩ argument is optional and specifies the debug level. Valid debug levels are 1, 2, 4, and 8, the amount of debug messages increases with the debug level. See the description of the -D argument for a way to restrict the amount of debug messages.
- The -D ⟨num⟩ argument is optional and restricts the debug messages produced during register allocation to those messages that concern the register with the specified number.
- The -s ⟨num⟩ argument is optional and specifies the statistics level. Valid levels are 1, 2, 4, 8, and 16, which produces a statistics summary after each program, module, procedure, super block, basic block, respectively.

Configuration Files. The assembler converter uses two configuration files to steer the code conversion process: the platform-specific configuration file and a module-specific configuration file. The module-specific configuration file is usually produced by the high-level language converter and contains

information about the internal and external procedures encountered in a given module. The structure of this configuration file has been described in Section 3.3.1.

The platform-specific configuration file is divided into three sections: The first section provides information about available instruction styles, i.e. the syntax of instructions and operands. The second section provides information about individual instructions using several instruction styles. The last section provides information about platform-specific system and library calls. The structure of the individual entries in the first two sections is described in the following paragraphs, the structure of the entries in the third section is identical to those in the module-specific configuration file.

An instruction style is a string that represents the instruction syntax and is determined by the number and type of operands. The individual entries in the first section of the platform-specific configuration file consist of the name and type fields: The type field consists of the type keyword followed by a non-negative integer and is used to assign a unique number to each instruction style, as these styles are later referenced by this number.

The name field consists of the format keyword followed by a string enclosed in quotes, which contains a representation of the instruction syntax. The ⟨exp⟩ and ⟨reg⟩ keywords are used to represent expressions and register names, respectively. For example, the syntax of an indirect addressing mode is represented by the following string:

```
"<reg>, <exp>(<reg>)"
```

The first ⟨reg⟩ keyword represents the name of the source or destination register, the ⟨exp⟩ keyword represents the offset, while the second ⟨reg⟩ keyword represents the name of the base register. Note that the first section must contain entries for all instruction styles that are used by entries in the second section, that is described below.

Each entry in the second section defines an instruction and style pair and consists of the name, type, info, qualifier, original, modified, and tune fields: The name field consists of the name keyword followed by a string enclosed in quotes that specifies the name of the instruction. The type field consists of the type keyword followed by an integer that specifies the instruction style. The number should reference one of the instruction styles that are defined in the first section of the configuration file.

The info field consists of the info keyword followed by a space-separated list of keywords and is used to provide semantic information about the instruction. The following keywords are supported:

- The CT_OPi keyword specifies that the ith operand of the instruction/style pair is a constant, i.e. a literal.
- The LD_OPi keyword specifies that the ith operand, which must be a register, of the instruction and style pair is read by the instruction.

- The ST_OPi keyword specifies that the ith operand, which must be a register, of the instruction and style pair is written by the instruction.
- The LD_FPCR, ST_FPCR keywords are platform-specific and specify that the instruction reads and writes the floating-point control register, respectively.
- The INT_x, FP_x keywords are used to group instructions into several sets as described in Section 3.5.9. This information is used for statistical purposes.
- The BR_CTRL keyword specifies that the instruction changes the control-flow, i.e. is a conditional, unconditional, or indirect branch.
- The BR_COND keyword is used in conjunction with the BR_CTRL keyword to specify that the instruction is a conditional branch.
- The BR_CALL keyword specifies that the instruction is a subroutine call, i.e. a type of indirect branch.
- The NOP keyword specified that the instruction is a null operation, i.e. performs no useful work.
- The TRAPB keyword specifies that the instruction is a trap barrier. This information is used in conjunction with the trapb optimization.
- The SWITCH keyword specifies that the instruction forces the end of an instruction block, thus causing a context switch.
- The SPECIAL keyword is used for some instructions that are only used by the assembler converter, e.g. procedure prologs.

The qualifier field consists of the qual keyword followed by a string enclosed in quotes. The string contains a list of supported instruction qualifiers for this instruction. An instruction qualifier is used as a postfix to an instruction and changes the behavior of the instruction, e.g. by enabling overflow checking or specific rounding modes.

The original field begins with the original keyword and is limited by curly brackets. Inside the brackets are one or more line fields. Each line field consists of the line keyword followed by a string enclosed in quotes, the style keyword followed by an integer as well as an optional tag keyword followed by one or more keywords. The string contains an instruction template that is used to print the original instruction. The template string contains several keywords that are substituted during the code conversion process:

- The ⟨qual⟩ keyword is substituted with the actual instruction qualifiers.
- The ⟨exp$i$⟩ keywords are substituted with the ith instruction operand, which must be a literal or expression.
- The ⟨Mreg$i$⟩ keywords are substituted with the reallocated register for the ith instruction operand, which must be a register.
- The ⟨Ooff$i$⟩ keywords are substituted with the offsets of the original register for the ith instruction operand, which must be a register. The offset is provided for special instructions that save or restore registers to or from the thread descriptor, respectively. The keyword denotes the offset of the corresponding register inside the descriptor structure.

- The ⟨Inum⟩ is substituted with the number of instructions that have been converted so far. This keyword is used to generate unique labels.

The style specifies the instruction style used by the template string and is required for proper processing of the template string. This field is similar to the type field and is used during the conversion process. The tag field is used to suppress individual lines under several conditions:

- The EMUFIX tag specifies that the corresponding line should be suppressed if the emufix optimization is enabled.
- The EMUVAR tag specifies that the corresponding line should be suppressed if the emufix optimization is disabled.
- The R*i*ZERO tag specifies that the corresponding line should be suppressed if the *i*th instruction operand is a zero source and sink register.

The modified field begins with the modified keyword and is limited by curly braces. Inside the brackets are one or more line fields similar to the ones described above. The only difference is the support for additional keywords in the line field:

- The ⟨Ireg*i*⟩ and ⟨Freg*i*⟩ keywords are substituted with the name of the *i*th temporary integer and floating-point register, respectively.
- The ⟨Moff*i*⟩ keywords are substituted with the offsets of the reallocated register for the *i*th instruction operand, which must be a register. The offset is provided for special instructions, that save or restore registers to or from the thread descriptor and denotes the offset of the corresponding register inside the descriptor structure.

The tune field is optional and provides implementation-specific informations about the instruction. The field begins with the tune keyword and is enclosed in curly braces. Inside the braces are three entries that specify the name of the implementation, the execution pipelines that can be used to execute the instruction, as well as the latency of the instruction. If the platform-specific configuration file contains multiple entries for the same instruction and style pair that only differ in their implementation-specific details, one entry is chosen according to the -O arch command-line argument.

3.5.2 Lexer & Parser

The assembler converter uses two passes to convert the assembler source: The first pass gathers platform-specific information and is therefore described in Section 3.7. The second pass identifies internal procedures and translates the corresponding instructions into an internal representation. Since the syntax of the assembler sources depends on the assembler, the second pass is usually platform-specific as well. Note that this holds even for platforms that use the same processor architecture. However, the platform-specific differences in the second pass concern only low-level details, the same tasks are performed on

all platforms. The two passes use a combination of lexer and parser for the lexical and syntactical analysis of the assembler source, which is described in the following paragraphs.

An assembler source file is a sequence of statements and comments which may cross multiple lines. Each statement is either an instruction, a directive, or a label. An instruction is an mnemonic as defined by the instruction set architecture followed by zero or more operands. Note that some assemblers define additional mnemonics for commonly used functions to ease the task of the assembler programmer. A directive is an instruction for the assembler itself. Directives are usually used for bookkeeping, storage reservation, and other control functions. Labels assign a name to a specific location in the program, such that branch instructions can reference the location by that name.

The lexer extracts the individual statements from the assembler source and divides each statement into individual tokens. These tokens represent the elements of a statement, e.g. mnemonics, constants, registers, operators, and punctuation. The individual tokens that make up a statement are passed to the parser, which handles the semantic analysis of the statement.

The assembler converter uses the lex program [LMB92] to construct the lexer for the assembler source. The lex program uses a specification of the individual tokens as regular expressions and produces a routine that is able to identify these tokens. This routine is automatically incorporated into the assembler converter. Note that the lexer must be able to distinguish between identifiers, instructions, directives, and register names in order to properly divide statements into tokens. Therefore the lexer maintains a list of all instructions, directives, and register names to identify these and separate them from the identifiers.

The parser processes the individual tokens from the lexer, checks the syntax of the corresponding statement, and performs statement-specific operations. The assembler converter uses the yacc program [LMB92] to construct a parser for the assembler source. The yacc program takes a grammar in Backus-Naur-Form, consisting of statements and associated actions, and turns it into a routine written in C. These routine is able to turn a sequence of tokens into a statement from the grammar and perform the corresponding actions.

The parser in the assembler converter operates as follows: By default, all statements are written to the output file until a directive is found that signals a procedure entry point. In this case the next label, i.e. the name of the procedure, is compared with the names of the internal procedures. If the procedure is external, the parser returns to the default mode. Otherwise, i.e. the procedure is internal, all statements are translated into an internal representation until a directive is found that signals the end of the procedure. After the whole procedure has been translated, the corresponding code is

converted and the parser examines the rest of the source file in order to identify other internal procedures.

3.5.3 Basic Blocks

A basic block is usually defined as a maximal sequence of instructions that can be entered only at the beginning and can be left only at the end of the sequence. The first instruction in a basic block is therefore either a branch target or an instruction that immediately follows a branch instruction. Likewise, the last instruction in a basic block is either a branch or an instruction that immediately precedes a branch target. Note that branch targets are usually denoted by labels, hence both terms are used interchangeably.

The assembler converter uses an extended definition of basic blocks: The length of the instruction sequence must be less than or equal to the grainsize g if a non-zero grainsize g was defined via the command-line. Calls to internal procedures are treated as branches in respect to the definition above, because these calls require a context switch. Recall from Section 2.3.1 that larger instruction blocks are beneficial to the performance of emulated multithreading. Therefore calls to external procedures are not treated as branches in order to increase the size of the instruction block.

The assembler converter uses a three-stage algorithm to create basic blocks: The first stage, illustrated in Figure 3.2, determines the size of the individual basic blocks. The second stage, illustrated in Figure 3.3, creates the actual basic blocks, while the third stage, illustrated in Figure 3.4, links the basic blocks according to the control-flow graph. The separation between these stages is not strictly necessary, but makes the implementation of the algorithm easier to debug and maintain.

The first stage processes all instructions in sequential order and marks those instructions that end a basic block. An instruction ends a basic block under one of the following conditions:

- The instruction is a branch, but not a call to an external procedure.
- The instruction always causes a context switch.
- The size of the current basic block is equal to the grainsize.
- The instruction immediately precedes a label.

Note that the last instruction in the procedure always ends a basic block.

The individual basic blocks are created during the second stage of the algorithm. The instructions are again processed in sequential order, adding instructions to the current basic block, until an end-of-bblock instruction is encountered. If the basic block has an associated label, this label is stored together with a pointer to the block for later reference during the third stage of the algorithm.

During the third stage, the individual basic blocks are linked according to the control-flow graph. The list of basic blocks created during the second stage is processed in the following way: The last instruction of every basic block is

Fig. 3.2. Creation of Basic Blocks - Stage I

```
while(inst_ptr = walk_list(proc_ptr->inst_list)) {

 if(inst_ptr->flag & INST_INSTRUCTION) {
   current_grainsize++;

   if((GET_BR_CTRL(inst_ptr) && !GET_BR_CALL(inst_ptr)) ||
      (GET_BR_CTRL(inst_ptr) &&  GET_BR_CALL(inst_ptr) &&
       GET_CTYPE_SWITCH(inst_ptr)) ||
      (current_grainsize >= max_grainsize) ||
      (inst_ptr->flag & INST_SWITCH)) {
    inst_ptr->flag    |= INST_END_BBLOCK;
    current_grainsize  = 0;
    }
   }
 else if(inst_ptr->flag & INST_LLABEL) {
  inst_ptr->prev_ptr->flag |= INST_END_BBLOCK;
  current_grainsize = 0; }
 }
```

Fig. 3.3. Creation of Basic Blocks - Stage II

```
bblock_ptr = new_bblock();

while(inst_ptr = walk_list(proc_ptr->inst_list)) {

 list_append(bblock_ptr->inst_list, inst_ptr);

 if(inst_ptr->flag & INST_LABEL) {
  list_append(bblock_ptr->labels_list, inst_ptr->name); }
 else if((inst_ptr->flag & INST_INSTRUCTION) &&
   (inst_ptr->flag & INST_END_BBLOCK)) {

   if(bblock_ptr->flag & BBL_NOP_ONLY) {
    merge_bblocks(proc_ptr->bblock_list.tail_ptr, bblock_ptr); }
   else {
    list_append(proc_ptr->bblock_list, bblock_ptr); }

   bblock_ptr = new_bblock();
   }
 }
```

Fig. 3.4. Creation of Basic Blocks - Stage III

```
while(bblock_ptr = walk_list(proc_ptr->bblock_list)) {
 inst_ptr = bblock_ptr->last_inst;

 if(GET_BR_CTRL(inst_ptr) && !GET_BR_CALL(inst_ptr)) {
  label = inst_get_label(inst_ptr);

  if(GET_BR_COND(info)) {
   link_bblocks(bblock_ptr, bblock_ptr->next_ptr); }

  link_bblocks(bblock_ptr, search_label(label));
  }
 else {
  link_bblocks(bblock_ptr, bblock_ptr->next_ptr); }

 }
```

inspected. If the instruction is not a branch or is an conditional branch, the current basic block is linked to the next basic block in the list. Note that the basic blocks are created, stored, and retrieved in sequential program order. If the last instruction is an conditional or unconditional branch, the branch target is looked up in the list of labels created during the second stage of the algorithm. The current basic block is then linked to the basic block that is associated with the label.

Lemma 3.5.1. *Let n be the number of instructions in the current procedure and $k \leq n$ be the number of basic blocks in the current procedure. The algorithm described above has a worst case runtime of $O(n \cdot k)$ or $O(n \log(k))$, depending on the type of data structure used to store the labels.*

Proof. The first stage of the algorithm processes all n instructions in the procedure and marks some of them as end-of-bblock instruction. Each instruction can be processed in constant time, hence the first stage of the algorithm has a worst-case runtime of $O(n)$.

Like the previous stage, the second stage processes all n instructions in the procedure. Instructions are added to the current basic block until an end-of-bblock instruction is encountered and is stored for later reference during the third stage. As the third stage references the basic blocks in the same order as they were stored in the second stage, a linked list is sufficient to store the individual blocks, hence insertion of a basic block takes constant time.

However, if a label is encountered, the label and a reference to the corresponding basic block are stored for later reference during the third stage. If a linked list is used to store the labels, the insertion takes constant time, but each search during the third stage will take linear time. Using more advanced

data structures, e.g. red/black trees, both insertion and searching take logarithmic time [CLR90]. Assume that every basic block has at most one label in order to determine the worst-case runtime of the second stage. Hence the worst-case runtime of the second stage is $O(k)$ or $O(k \log(k))$, depending on the data structure used to store the labels.

The third stage processes all basic blocks and creates links between the basic blocks according to the control-flow graph. It takes constant time to link two basic blocks, but determining the branch target requires a search for the corresponding label. As mentioned above, each search takes linear or logarithmic time, depending on the data structure that is used to store the labels. In the worst case, each basic block requires a search of this data structure, i.e. every basic block ends with a conditional branch instruction. As there are at most n basic blocks, the worst-case runtime of the third stage is $O(n \cdot k)$ or $O(n \log(k))$, which is also the overall runtime. □

In practice, the number k of basic blocks will be much smaller than n, i.e. $k \ll n$. The assembler converter uses a linked list to store the labels, since the runtime of basic block creation is insignificant compared to other parts of the converter, e.g. register allocation.

3.5.4 Super Blocks

The size of the instruction blocks has a significant impact on the performance of emulated multithreading: The larger the instruction blocks, the fewer iterations of the main loop in the thread execution routine are required, especially if the instruction blocks are large enough to contain whole loops.

Basic blocks, as introduced in Section 3.5.3, are not well-suited as instruction blocks, since they are often too small: On the average every 6th instruction is a branch, which limits the average size of basic blocks to six instructions [Wal92]. Optimizing compilers use several techniques to extend the size of basic blocks, e.g. loop unrolling and trace scheduling, thereby providing benefits to emulated multithreading. However, instruction blocks should extend across loops in order to decrease the number of iterations in the main loop of the thread execution routine. An instruction block should therefore consist of several basic blocks. These sets of basic blocks must satisfy the following conditions:

- The control-flow graph induced by the sets of basic blocks should form a subgraph of the control-flow graph of the program, i.e. two of these sets are connected if and only if two basic blocks from the corresponding sets are connected in the control-flow graph. This condition ensures that the program semantics are maintained.
- The set of basic blocks should have a single point of entry, i.e. there are no edges from basic blocks outside the set to other basic blocks inside the set in the control-flow graph. This condition ensures that all basic blocks in

the set are reachable via the entry point. This property is exploited during the creation of super blocks and simplifies the integration of context switch code.
- The set must not extend across designated basic blocks. If context switches are performed at each exit point of the set, this condition ensures that context switches are performed after these basic blocks.

As there is no restriction on the number of exit points, i.e. such a set must have a single entry point, but may have multiple exit points. Structures with similar properties have been published in literature, e.g. extended basic blocks, super blocks, minimum and maximum intervals. The following paragraphs describe these structures in detail and discuss their merits with respect to emulated multithreading.

An extended basic block [Muc97] is a maximal sequence of basic blocks such that only the basic block at the beginning can have more than one predecessor. The extended basic blocks of a control-flow graph is constructed using a depth-first-search algorithm in $O(n+m)$ time, where n is the number of basic blocks and m is the number of edges in the flow graph [Muc97]. Extended basic blocks meet the first two conditions from above, but can only include a single loop.

Super blocks are structures that are based on trace scheduling: Trace scheduling divides the flow graph into a set of traces that represent the frequently executed paths in the flow graph [Fis81]. Traces may contain side exits, i.e. branches out of the trace as well as side entries, i.e. branches from other traces into the middle of the trace. A super block is a trace that has no side entrances [HMC+93]. Super blocks are formed in two steps: First, traces are identified using either profiling information [HMC+93] or static program analysis [HMB+93]. Second, tail duplication [Fis81] is performed to eliminate any side entrances to the trace. A related structure is the hyper block, which uses predication to integrate multiple paths of control in the same hyper block [MLC+92]. The super and hyper blocks meet the first two conditions from above, but cannot include loops, since all side entrances, even those from the trace itself, are removed.

Intervals are used for control-flow analysis and come in two forms: maximal and minimal intervals [Muc97]. A maximal interval with header h is the maximal, single entry subgraph of the control-flow graph, such that h is the only entry node and all closed paths in the subgraph contain h. A minimal interval is defined to be either a natural loop, a maximal acyclic subgraph or a minimal irreducible region. A natural loop of a back edge (p, q) in the control-flow graph, where p dominates q, is the subgraph consisting of the set of nodes containing q, all the nodes from which p can be reached without passing through q, as well as the corresponding edges. Note that both types of intervals allow only a single loop per interval.

The above approaches are too restricted for the purpose of emulated multithreading, hence a new structure called a super block[1] is defined: A super block is a maximal set of basic blocks that meets the conditions presented above. Apart from the conditions, there are no further restrictions. The designated basic blocks that limit the size of these super blocks are called end-of-sblock blocks, while the entry block in in a super block is called start-of-sblock. The assembler converter uses super blocks as instruction blocks if the corresponding optimization is enabled, otherwise basic blocks are used. The algorithm used to identify the super blocks is based on depth-first search (DFS) and is described in the following paragraphs.

Algorithm. The main routine of the algorithm is depicted in Figure 3.5. The algorithm maintains a list of super block headers, i.e. entry points to a new super block, initialized to the entry point of the procedure. Note that all basic blocks in the procedure are reachable via the procedure entry point.

The first and last step in the algorithm reset the level information for all basic blocks that belong to the current procedure. This information is used during the discovery of new super blocks as well as some of the other algorithms in the assembler converter. The main loop is executed until all basic blocks have been processed. Inside the loop, the first entry in the header list is removed from the list and a new super block is created from this entry. Afterwards links to predecessor super blocks are created for all predecessor basic blocks that are already part of a super block. The remaining links will be completed during the later stages.

Three subroutines are used to extend the current super block, the following paragraphs describe these three subroutines in detail.

The first subroutine, illustrated in Figure 3.6, is based on the visit stage of depth-first search and visits all basic blocks that are reachable from the current header. Upon entry to a basic block, the following operations are performed: If the level of the basic block is different from the current level, the level of the basic block is set to the current level and the number of visits is cleared. This is necessary because the number of visits may still contain information from previous traversals of the control-flow graph. Afterwards the number of visits is incremented. If the basic block is visited for the first time and is not an end-of-sblock block, all children of the basic block are visited recursively. The end-of-sblock condition ensures that the third condition above is met.

The first subroutine differs in two ways from the visit stage of the depth-first search algorithm: The initialization of the number of visits in case of left-over information, and the additional check of the end-of-sblock condition before the children are visited recursively.

The second subroutine, illustrated in Figure 3.7, is based on the visit stage of the depth-first search algorithm as well. The subroutine is used to determine the actual size of the super block and updates the end-of-sblock,

[1] The name was chosen independently of [HMC+93].

Fig. 3.5. Creation of Super Blocks - Main

```
int level = 1;

while(bblock_ptr = walk_list(proc_ptr->bblock_list)) {
 bblock_ptr->level = 0; }

list_init(&header_list);
list_append(&header_list, proc_ptr->bblock_list.head_ptr);

while(!list_empty(header_list)) {

 bblock_ptr = list_remove(header_list, header_list.head_ptr;);
 sblock_ptr = new_sblock();

 while(bblock_ptr = walk_list(bblock_ptr->parent_list))
  if(bblock_ptr->sblock_ptr)
   link_sblocks(bblock_ptr->sblock_ptr, sblock_ptr);

 visit_sblock(bblock_ptr, level);

 size_sblock(sblock_ptr, bblock_ptr);

 fill_sblock(sblock_ptr, bblock_ptr);

 level++;
 }

while(bblock_ptr = walk_list(proc_ptr->bblock_list)) {
 bblock_ptr->level = 0; }
```

Fig. 3.6. Creation of Super Blocks - Stage I

```
void visit_sblock(struct bblock *bblock_ptr, int level)
{

if(bblock_ptr->level != level) {
 bblock_ptr->num_visits = 1;
 bblock_ptr->level      = level;

 if((bblock_ptr->num_visits == 1) &&
   !(bblock_ptr->flag & BBL_END_SBLOCK))
   while(child_ptr = walk_list(bblock_ptr->childs_list)) {
    visit_sblock(child_ptr, level); }
 }
else {
 bblock_ptr->num_visits++; }

}
```

Fig. 3.7. Creation of Super Blocks - Stage II

```
static void size_sblock(struct sblock *sblock_ptr,
                        struct bblock *bblock_ptr)
{
if(bblock_ptr->level != 0) {
 bblock_ptr->level = 0;

 while(child_ptr = walk_list(bblock_ptr->childs_list)) {

  if(!(bblock_ptr->flag & BBL_END_SBLOCK)) {
   if(childs_ptr->sblock_ptr) {
    if(childs_ptr->sblock_ptr != sblock_ptr) {
     bblock_ptr->flag             |= BBL_END_SBLOCK; }
    }
   else {
    if(childs_ptr->num_visits ==
       childs_ptr->parent_list.num_elements) {
     if(!(childs_ptr->flag & BBL_NEW_SBLOCK)) {
      size_sblock(sblock_ptr, childs_ptr); }
     else {
      bblock_ptr->flag            |= BBL_END_SBLOCK; }
     }
    else {
     bblock_ptr->flag             |= BBL_END_SBLOCK;
     childs_ptr->flag             |= BBL_NEW_SBLOCK; }
    }
   }
  else if(childs_ptr->sblock_ptr == NULL) {
    childs_ptr->flag             |= BBL_NEW_SBLOCK; }
  }
 }
}
```

start-of-sblock flags accordingly. Upon entry to a basic block, the following operations are performed:

The subroutine checks the level to ensure that each basic block is processed only once. If the level information is already zero, the basic block has been processed before and the subroutine returns immediately. Otherwise the level is set to zero and all children of the basic block are examined. If the basic block is an end-of-sblock basic block and the current child is not part of any super block, the child is marked with the start-of-sblock flag. This child will be added to the header list by the third subroutine.

Otherwise the child is examined in the following way: If the child already belongs to a different super block, the current basic block is marked with an end-of-sblock flag and the subroutine returns. If the child does not belong to any super block, the number of visits is compared with the number of predecessors of the child. If both numbers are equal, the child can be added

Fig. 3.8. Creation of Super Blocks - Stage III

```
static void fill_sblock(struct sblock *sblock_ptr,
                        struct bblock *bblock_ptr)
{

bblock_ptr->sblock_ptr  = sblock_ptr;
list_append(&sblock_ptr->bblock_list, bblock_ptr);

while(child_ptr = walk_list(bblock_ptr->childs_list)) {

 if(!(bblock_ptr->flag & BBL_END_SBLOCK)) {
  if(child_ptr->sblock_ptr) {
   if(child_ptr->sblock_ptr != sblock_ptr) {
    link_sblocks(sblock_ptr, child_ptr->sblock_ptr); }
   }
  else {
   if(child_ptr->flag & BBL_NEW_SBLOCK) {
    if(search_bblock(header_list, child_ptr) == NULL) {
     list_append(header_list, child_ptr); }
    }
   else {
    fill_sblock(sblock_ptr, childs_ptr); }
   }
  }
 else {
  if(child_ptr->sblock_ptr) {
   link_sblocks(sblock_ptr, child_ptr->sblock_ptr); }
  else {
   if(search_bblock(header_list, child_ptr) == NULL) {
    child_ptr->flag |= BBL_NEW_SBLOCK;
    list_append(&header_list, child_ptr); }
   }
  }
 }
}
```

to the super block without violating the single-entry property. However, if the child is marked with the start-of-sblock flag, it cannot be added to the current super block, hence the basic block is marked with the end-of-sblock flag. Otherwise the second subroutine is recursively called for the child in order to explore other basic blocks that may be added to the super block.

If the number of visits is not equal to the number of predecessors, the current child cannot be added to the super block, hence the basic block is marked with the end-of-sblock flag, while the current child is marked with the start-of-sblock flag. The child will be added to the header list by the third subroutine.

Like the previous two subroutines, the third subroutine, illustrated in Figure 3.8, is based on the visit stage of the depth-first search algorithm. This

subroutine adds the individual basic blocks to the super block and updates the header list. Upon entry to a basic block, the following operations are performed:

The basic block is added to the current super block and all children of the block are examined similar to the second subroutine: If the current basic block is marked with the end-of-sblock flag, it is checked whether the child belongs to another super block. If the child already belongs to another super block, the corresponding links between these two super blocks are created. Otherwise the child is marked with the start-of-sblock flag and added to the header list, if it is not already present.

If the current basic block is not marked with the end-of-sblock flag, it is checked whether the current child belongs to another super block. If the child already belongs to another super block, the corresponding links between these two super blocks are created. If the child does not belong to another super block, but is marked with the start-of-sblock flag, it is added to the header list, if it is not already present. Otherwise, the child is added to the super block by recursively calling the third subroutine.

The assembler converter uses the algorithm described above to create the individual instruction blocks. Recall that these instruction blocks must meet the three conditions described above, i.e. the single-entry, subgraph, and end-of-sblock conditions. The correctness of the algorithm as well as the worst case runtime for two different versions of this algorithm is proven in the remainder of this section.

Lemma 3.5.2. *The super blocks created by the algorithm above have a single point of entry.*

Proof. This property is proven by contradiction: Given two different super blocks s_1, s_2 with headers h_1, h_2, assume that an edge (u, v) exists, such that the nodes u, v belong to super blocks s_1, s_2, respectively and $v \neq h_2$ holds. Since all basic blocks in the flow graph are reachable by the procedure entry point h, there exists at least one path from h to v. This path does not contain h_2, otherwise u and v would be in the same super block: The first subroutine visits all basic blocks that are reachable from h_2 and updates the number of visits accordingly. After completing the first stage, the number of visits of v is smaller than the number of predecessors of v, since a path from h to v exists that does not contain h_2. Node v can only be added to the super block s_2 if both numbers are equal, hence h_2 does not belong to s_2. This contradiction completes the proof. □

Lemma 3.5.3. *The super blocks created by the algorithm above are disjoint and cover the whole control-flow graph.*

Proof. If a basic block is added to a super block, that basic block will neither be added to another super block nor to the header list. Hence, the created super blocks are disjoint.

In the flow graph, each basic block is reachable from the entry point, and is either placed in the header list or added to a super block. Unless a basic block is added to a super block, it will be the header of another super block. Therefore the union of all super blocks covers the whole flow graph. □

Lemma 3.5.4. *The links between the super blocks form an abstract flow graph.*

Proof. This property is proven by contradiction: Given two super blocks s_1, s_2 and basic blocks b_1, b_2 that belong to s_1, s_2, respectively, assume that (b_1, b_2) are linked in the flow graph induced by the basic blocks, while (s_1, s_2) are not linked in the flow graph induced by the super blocks.

Without loss of generality, assume that b_1 was created before b_2. b_2 cannot be the header of s_2, otherwise (s_1, s_2) would have been linked in the main routine, since s_1 was already present at that time. On the other hand, b_2 cannot be any other block in s_2, otherwise (s_1, s_2) would have been linked by the third subroutine during the creation of s_2. Therefore b_2 cannot belong to s_2, which is a contradiction. □

Theorem 3.5.1. *The worst case runtime of the super block algorithm as described above is $O(n(n+m))$, where n is the number of basic blocks and m is the number of edges in the abstract flow graph induced by the basic blocks.*

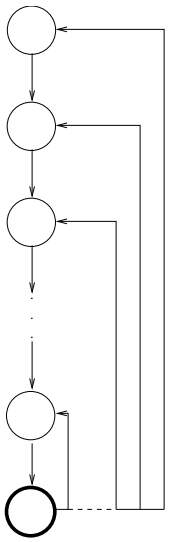
Proof. The first and last stages in the algorithm, i.e. the clearing of the level information for all basic blocks, have a worst-case runtime of $O(n)$.

The first subroutine is executed for every header block and visits all basic blocks that are reachable from the header block without crossing end-of-sblock boundaries. In the worst case, every basic block is a header, hence there are at most n headers. It is possible to construct a flow graph that has n headers and almost all n basic blocks have to be visited for each header, i.e. a full depth-first-search has to be performed. Such a flow graph is depicted in Figure 3.9. Note that only the last basic block is marked with the end-of-sblock flag. A single depth-first search has a worst-case runtime of $O(n+m)$, hence the first stage of the super block algorithm has a worst-case runtime of $O(n(n+m))$.

The second stage of the algorithm is executed for all header blocks. For each header, all basic blocks that will be added to the corresponding super block in the third stage are visited. Since the individual super blocks do not overlap, each node in the flow graph is visited and each edge is traversed exactly once. The worst-case runtime of the second stage is therefore identical to the worst-case runtime for depth-first search, which is $O(n+m)$.

The third stage of the algorithm is executed for each header block. For each header block, all basic blocks that belong to the corresponding super block are visited. Using the same argument as above, the worst-case runtime for the third stage of the algorithm is $O(n+m)$. □

Fig. 3.9. Example for worst-case Control-Flow Graph



The overall worst-case runtime algorithm is dominated by the worst-case runtime of the first stage, i.e. $O(n(n+m))$. This result can be significantly improved by changing the first stage of the algorithm as described below.

Theorem 3.5.2. *The worst-case runtime of the super block algorithm can be improved to $O(n+m)$, where n is the number of basic blocks and m is the number of edges in the abstract flow graph induced by the basic blocks.*

As described above, the first stage of the algorithm visits some basic blocks unnecessarily, i.e. nodes that can never be part of the corresponding super block. In order to avoid these visits, the notion of dominance is useful: Given a flow graph $G = (V, E)$ with entry point r, a vertex u is said to be dominated by another vertex v, if every path from r to u contains v. The following lemma provides the connection between dominance and super blocks.

Lemma 3.5.5. *All basic blocks in an super block are dominated by the header.*

Proof. This lemma can be proven by contradiction: Given a super block s with header h, assume that a basic block b in s exists, such that h does not dominate b. If b is not dominated by h, there exists at least one path from the entry point r to b that does not contain h. This contradicts the single-entry property proven in Lemma 3.5.2. □

Proof. Based on the above lemma, the algorithm is changed by computing all dominators for all basic blocks in the abstract flow graph and changing the first stage in the following way: A child is only visited if it is dominated by the header of the current super block. This modification reduces the worst-case runtime of the first stage to $O(n + m)$, since the first stage now visits all basic blocks and traverses all edges in the flow graph exactly once.

The dominance for all nodes in a flow graph can be determined in near-linear time, i.e. $O(m\alpha(m, n))$, where $\alpha(m, n)$ is a functional inverse of Ackermanns function, i.e. an extremely slow growing function [LT79]. □

3.5.5 External Calls

Emulated multithreading distinguishes two types of procedures, i.e. internal and external procedures. An internal procedure is a procedure that uses emulated multithreading, i.e. the corresponding code is modified during the code conversion process. An external procedure is a procedure that does not use emulated multithreading, i.e. the corresponding code is not modified during the code conversion process.

The distinction between internal and external procedures has two advantages: First, emulated multithreading can be applied on a procedure-by-procedure basis, i.e. restricted only to those procedures that benefit from emulated multithreading. Second, calls to procedures for which no source code is available, e.g. system calls, can be executed. However, calls to external procedures must follow the standard calling conventions, which complicates the code conversion process.

The following paragraphs describe the approach used by the assembler converter to handle external calls: Based on a general description of procedure calls, implications for external calls in combination with emulated multithreading are derived. Two solutions that address the corresponding problems are presented, a simple and a complex one. The latter solution is described in detail, including the algorithmic implementation.

Procedure Calls. A procedure call can be divided into three parts: the call prolog, the actual call, and the call epilog. The call prolog assembles the arguments and stores them in the corresponding argument registers or stack locations according to the calling convention. Apart from assembling the arguments, the call prolog is also responsible for saving any callee-save registers to memory as well as calculating the address of the callee. The actual call transfers control to the callee and saves the return address in a register. The call epilog handles the return values and restores any register that may have been destroyed during the call. The calling conventions are platform- as well as operating-system-specific.

An example for a procedure call is depicted in Figure 3.10. This example was taken from a program compiled for the Alpha architecture under the Tru64 operating system. The semantics of the individual instructions and addressing modes are described in Appendix A.

Fig. 3.10. Procedure Call Example

```
ldq     $27, getopt($gp)!literal!11
mov     $9,  $16
mov     $10, $17
jsr     $26, ($27), getopt!lituse_jsr!11
ldah    $gp, ($26)!gpdisp!13
lda     $gp, ($gp)!gpdisp!13
```

According to the Tru64 calling conventions [Tru96] the integer registers, i.e. r0-r31, are used in the following way: Arguments are passed in registers r16-r21, return values are returned in register r0. Register r26 holds the return address, while r27 holds the procedure value, i.e. the address of the callee. Register r28 is reserved for the assembler and registers r29, r30 hold the global and stack pointers, respectively. The remaining registers are temporary registers that are used for expression evaluation. Note that only registers r9-r15 and r26 are callee-save, all other registers are not preserved across procedure calls.

The call prolog ranges from the ldq instruction to the mov instruction, the actual call is performed by the jsr instruction. The call epilog consists of the ldah, lda instruction pair after the actual call. The address of the procedure is assembled in register r27 by the lda/ldah instruction pair. This register is used in the actual call, i.e. the jsr instruction, to provide the address of the callee. The arguments are assembled in register r16 and r17, register r30 contains the stack pointer. The call epilog restores the global pointer after the call, since the register used for the global pointer is not callee-save. The procedure return value is returned in register r0.

Procedure calls present a problem for emulated multithreading: On the one hand, the called procedure expects arguments in specified registers, on the other hand all registers are reallocated during the code conversion process. If both the caller and the callee use emulated multithreading, the problem does not exist: The entry point of the callee is the header of a super block by definition. Therefore all internal procedures can only be called via a context switch, such that the basic block that contains the call ends the corresponding super block. Hence the caller automatically saves all modified registers to the thread descriptor, while the callee loads all register values from the same descriptor after one or more context switches. Both the caller and the callee are free to reallocate all registers and ignore the calling conventions, as these conventions are preserved via the context stored in the thread descriptor.

The situation is different for external calls, since the callee further expects the register at the locations specified by the calling conventions. This problem can be solved by saving all modified registers to the thread descriptor prior to the call and inserting instructions that restore all registers needed by the call from the thread descriptor to the registers specified by the calling

conventions. In the same way, instructions to save the return value to the thread descriptor have to be inserted after the call. Otherwise the modified code that follows would use outdated values for these registers.

This solution is easy to implement once the registers that are needed for an individual call are known. Note that this information is already provided in one of the configuration files, as described in Section 3.3.1. Therefore the assembler converter only has to identify calls to external procedures and insert the corresponding save and restore instructions. However, this solution is not very efficient, as it causes a lot of accesses to the thread descriptor. Most of these accesses are unnecessary, because the call prolog already uses all of these registers.

Instead of the simple solution presented above, the assembler converter uses a more sophisticated solution: Inside the call prologue and epilogue, all registers that are needed by the call are allocated to themselves, hence the calling conventions are preserved. However, this solution requires the identification of the call prologue and epilogue, a complex task if no informations beside the assembler source is available. The algorithm used by the assembler converter is described in the following paragraphs.

The algorithm processes all basic blocks in the procedure. For each basic block, all instructions are processed in sequential order. If a procedure call instruction is encountered, the type of the called procedure is determined: In the case of internal procedures, a platform-specific routine is executed for bookkeeping reasons. In the case of external procedures, the register mask c_1 is created that contains all argument registers needed by the called procedure. If the call uses a fixed number of arguments, this register mask is obtained from one of the configuration files.

For procedures with a variable number of arguments, a heuristic is used to create the call mask: Beginning with the call instruction, all instructions are examined in reverse order, until an instruction is found that has more than one predecessor. All argument registers that are written by one of these instructions are recorded in the register mask. This heuristic is usually sufficient to find all argument registers that a procedure expects. It is possible to construct a flow graph that causes this heuristic to fail. However, the assembler converter operates on assembler code that was generated by the compiler, i.e. the generated code is "well-behaved", such that this heuristic works in practice. This situation can be resolved if the assembler converter is integrated into the compiler, since the compiler already has the required information.

After all argument registers have been determined, the call mask c_1 is updated with the mask of optional registers, which is obtained from one of the configuration files. A second call mask c_2 is created that contains all registers modified by the call instruction as well as the required registers. Taken together, the two call masks represent all registers that are expected

by the callee. The first mask is used to identify the call prologue as described below, the second call mask is used afterwards to update the first call mask.

The call prologue is identified by processing all instructions in reverse order, starting with the call instruction. Each instruction is examined and all registers that are written by one of these instructions are recorded in the c_3 live mask. The processing continues until all registers from the c_1 call mask have been recorded, another call prologue, epilogue, or an end-of-sblock block is encountered. In the latter two cases, the call prologue reaches from the call instruction to the last instruction that made a useful contribution to the c_3 mask, i.e. writes a register that is present in the c_1 mask, but has not been written by one of the previous instructions. The simple solution presented above is used for the remaining registers, i.e. these registers are reloaded just before the call.

All instructions in the call prologue are marked accordingly and the combined c_1 and c_2 call masks are stored in each instruction for later reference during register allocation. Inside a call prologue or epilogue, the register allocator will allocate all registers present in these masks onto themselves.

Note that the algorithm crosses basic block boundaries as long as the basic blocks are sequential, i.e. the current basic block has only one predecessor. These basic blocks as well as the corresponding super blocks are subsequently merged. The merging of super blocks is only possible if the super block optimization is disabled, i.e. super blocks are identical to basic blocks: The algorithm presented above would have added these basic blocks to the same super block. As a consequence, the final shape of basic and super blocks is only known after all external calls have been processed. For this reason, the original statistics as well as the pre- and post-order traversals of the abstract flow graph induced by the basic blocks are created only after the processing of external calls has been completed.

3.5.6 Data-Flow Analysis

Data-flow analysis is used to gather information about the way data is manipulated within a program. This information is a prerequisite for most optimization passes in modern compilers. As such, data-flow analysis must provide information that is accurate enough to enable optimizations, yet is conservative enough to prevent the optimization from changing the program semantics.

There are two different forms of data-flow analysis: inter-procedural analysis and intra-procedural analysis. Inter-procedural data-flow analysis is concerned with the flow of data between the individual procedures in a program, while intra-procedural analysis is concerned with the flow of data within procedures. The remainder of this section covers intra-procedural analysis, although the theoretical concepts apply to inter-procedural analysis as well.

One of the most important data-flow problems is the determination of reaching definitions and live variables:

Definition 3.5.1. *A definition of a variable, i.e. an assignment, is reaching at a given point in the procedure, if a path in the control-flow graph exists, such that the variable may still have the assigned value at that point. In this case, the goal of data-flow analysis is to determine the reaching definitions at all nodes in the control-flow graph.*

Definition 3.5.2. *A variable is live at a given point in the procedure, if a path in the control-flow graph from that point to an exit point exists, such that this path contains an use, i.e. a reference, of this variable. In this case, the goal of data-flow analysis is to determine the live variables at all nodes in the control-flow graph.*

Data-flow problems can be grouped into three different classes based on the direction of the information flow: forward flow, backward flow, and bidirectional problems. Forward flow problems process information in the direction of program flow, backward flow problems process information in the opposite direction. Bidirectional problems process information in both directions, but are rare in practice.

The remainder of this section introduces the theoretical foundations for data-flow analysis as well as a simple iterative algorithm to solve data-flow problems. The algorithm solves data-flow problems under certain conditions, the example problems based on the two definitions above are shown to satisfy these conditions. These results will be used to introduce the new data-flow analyses required to determine the shape of the individual live ranges prior to register allocation.

Lattice Theory. Lattice theory provides the foundation for data-flow analysis. Data-flow analysis is performed on elements of a structure called semi-lattice:

Definition 3.5.3. *A semi-lattice is a pair $(S, \sqcup)$, where S is a non-empty set and $\sqcup$ is a binary operation on S that is idempotent, commutative, and associative:*

$$\begin{aligned} \sqcap : S \times S &\to S \\ x \sqcup x &= x && \forall x \in S \\ x \sqcup y &= y \sqcup x && \forall x, y \in S \\ x \sqcup (y \sqcup z) &= (x \sqcup y) \sqcup z && \forall x, y, z \in S \end{aligned}$$

Either the $\sqcap$ or the $\sqcup$ symbols are used for the binary operation: In the former case, the operation is called join, in the latter case, the operation is called meet. This nomenclature is based on the definition of a larger structure called a lattice:

Definition 3.5.4. *A lattice is a triple $(S, \sqcap, \sqcup)$, where S is a non-empty set and $(S, \sqcap)$ and $(S, \sqcup)$ are semi-lattices.*

An example for a lattice is the set of m-bit vectors with the logical and ($\wedge$) and or ($\vee$) bit operations as the meet and join operations, respectively. Since there are 2^m different m-bit vectors, the set is non-empty. The fact that the logical bit operations are idempotent, commutative, and associative can be derived from the corresponding properties of the boolean and/or operations.

Given a lattice $(S, \sqcap, \sqcup)$, the meet and join operations induce partial orderings on the set S, denoted by $\sqsubseteq, \sqsupseteq$ for the join and meet operations, respectively:

$$x \sqsubseteq y \Leftrightarrow x \sqcap y = x$$
$$x \sqsupseteq y \Leftrightarrow x \sqcup y = x$$

Based on the above definition, the operations $\sqsubset$, $\sqsupset$ and $\sqsupseteq$ are defined correspondingly. Note that all operations can be defined either in terms of the meet or in terms of the join operation.

Lemma 3.5.6. *The $\sqsubseteq$ operation is reflexive, symmetric, and transitive:*

$$\begin{array}{rl} x \sqsubseteq x & \forall x \in S \\ x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y & \forall x, y \in S \\ x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z & \forall x, y, z \in S \end{array}$$

Proof. The reflexivity follows directly from the fact that $\sqcap$ is idempotent. The symmetry follows from the definition of the $\sqsubseteq$ operation and the fact that $\sqcap$ is commutative:

$$\left.\begin{array}{l} x \sqsubseteq y \Leftrightarrow x \sqcap y = x \\ y \sqsubseteq x \Leftrightarrow y \sqcap x = x \sqcap y \end{array}\right\} \Leftrightarrow x = y$$

The transitivity follows from the definition of the $\sqsubseteq$ operation and the associativity of $\sqcap$:

$$\left.\begin{array}{l} x \sqsubseteq y \Leftrightarrow x \sqcap y = x \\ y \sqsubseteq z \Leftrightarrow y \sqcap z = y \end{array}\right\} \Leftrightarrow x \sqsubseteq z$$

□

A (semi-)lattice can have two unique elements, the bottom element $\bot$ and the top element $\top$, that satisfy the following conditions:

$$\begin{array}{ll} x \sqcap \bot = \bot & \forall x \in S \\ x \sqcup \top = \top & \forall x \in S \end{array}$$

A (semi-)lattice is said to be of finite length, if every strictly increasing chain of elements

$$\bot = x_1 \sqsubset x_2 \sqsubset \ldots \sqsubset x_n = \top$$

is finite. Obviously, if S is finite the semi-lattice is of finite length.

Definition 3.5.5. *Let $L = (S, \sqcup)$ be a semi-lattice of finite length with a bottom element. A set F of functions on S is a monotone operation space associated with L, if and only if each $f \in F$ is monotone, F contains the identity operation , F is closed under composition, and F is complete [Hec77]:*

$$\begin{array}{rll} \forall f \in F : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y) & & \forall x, y \in S \\ \exists e \in F : e(x) = x & & \forall x \in S \\ \forall f, g \in F : f \circ g(x) = f(g(x)) \in F & & \forall x \in S \\ \forall x \in S : \exists f \in F : x = f(\bot) & & \end{array}$$

A monotone operation space associated with L is called distributive if the functions in F are distributive under the meet or join operation:

$$\forall f \in F : f(x \sqcup y) = f(x) \sqcup f(y) \qquad \forall x, y \in S$$

Note that the last condition implies that all functions in F are monotone:

$$x \sqsubseteq y \Leftrightarrow x \sqcup y = x \Leftrightarrow f(x) = f(x \sqcup y) = f(x) \sqcup f(y) \Leftrightarrow f(x) \sqsubseteq f(y)$$

Definition 3.5.6. *A monotone data-flow analysis framework is a triple $D = (S, \sqcup, F)$ such that $(S, \sqcup)$ is a semi-lattice of finite length with a bottom element and F is a monotone operation space associated with the semi-lattice $L = (S, \sqcup)$. A distributive data-flow analysis framework is a monotone data-flow analysis framework $D = (S, \sqcup, F)$ where F is distributive.*

The connection between a data-flow analysis problem and the previous definitions is provided by the following definition:

Definition 3.5.7. *An instance I of a monotone data-flow analysis framework $D = (S, \sqcup, F)$ is a tuple $I = (G, M)$, such that $G = (N, E, s)$ is a flow graph with entry point s and $M : N \to F$ maps each node in N to a function in F.*

The two examples presented above, i.e. reaching definitions and live variables, can be modeled as distributive data-flow analysis frameworks: In both cases, let S be the set of m-bit vectors and let $\sqcup$ be the logical or operation. As mentioned above, $(S, \sqcup)$ is a semi-lattice. Since the number of elements in S is 2^m, i.e. finite, each strictly increasing chain of elements from S is finite, hence the semi-lattice is of finite length. The bottom element $\bot$ of S is represented by the bit vector of all zero, since

$$x \sqcup \bot = x \vee (0, \ldots, 0) = x$$

Let F be the set of functions $\langle k, g \rangle$, such that

$$\langle k, g \rangle(x) = ((x \wedge \overline{k}) \vee g)$$

Lemma 3.5.7. *The triple $REACH = (S, \sqcup, F)$ is a distributive data-flow analysis framework.*

Proof. The monotonicity will be handled at the end. The identity operation in F is given by $\langle 0, 0\rangle$:

$$\langle 0, 0\rangle(x) = ((x \wedge \overline{0}) \vee 0) = x \qquad \forall x \in S$$

Let $\langle k_1, g_1\rangle$, $\langle k_2, g_2\rangle$ be two functions from F. The composition of these functions is a function in F:

$$\begin{aligned}\langle k_1, g_1\rangle(\langle k_2, g_2\rangle(x)) &= \langle k_1, g_1\rangle((x \wedge \overline{k_2}) \vee g_2) \\ &= (((x \wedge \overline{k_2}) \vee g_2) \wedge \overline{k_1}) \vee g_1 \\ &= ((x \wedge (\overline{k_2} \wedge \overline{k_1})) \vee (g_2 \wedge \overline{k_1} \vee g_1) \\ &= \langle(\overline{k_2} \wedge \overline{k_1}), (g_2 \wedge \overline{k_1} \vee g_1)\rangle\end{aligned}$$

For $x \in S$, the function $\langle 0, x\rangle$ meets the condition:

$$\langle 0, x\rangle(0) = (0 \wedge \overline{0}) \vee x = x$$

To show that F is distributive, let $x, y \in S$ and $f = \langle k, g\rangle \in F$. For every $1 \leq i \leq m$, two cases must be considered:

- Suppose the ith bit of $x \sqcup y$ is zero, i.e. the ith-bits of both x and y are zero. Therefore the ith bit of $f(x), f(y)$ equals the ith bit of g, as

  $$\begin{aligned}f(x) = \langle k, g\rangle(x) = ((x \wedge \overline{k}) \vee g) \\ f(y) = \langle k, g\rangle(x) = ((y \wedge \overline{k}) \vee g)\end{aligned}$$

  It follows that the ith bit of $f(x) \sqcup f(y)$ equals the ith bit of g, which equals the ith bit of $f(x \sqcup y)$ by the same reasoning, as the ith bit of $x \sqcup y$ was supposed to be zero.
- Suppose the ith bit of $x \sqcup y$ is one, i.e. at least one of the ith bits of x, y is one. Without loss of generality, assume that the ith bit of x is one. Then the ith bit of

  $$f(x) = ((x \wedge \overline{k}) \vee g) = \overline{k} \vee g \ ,$$

  which equals the ith bit of $f(x) \sqcup f(y)$. Applying the same reasoning to $f(x \sqcup y)$ yields that the ith bit of $f(x \sqcup y)$ is $\overline{k} \vee g$.

Note that the distributiveness of F implies the monotonicity.

Note that the triple $\text{LIVE} = (S, \sqcup, F)$ is a distributive data-flow analysis framework that uses the same semi-lattice $(S, \sqcup)$ and the associated operation space F and solves the live variables problem. The only difference between the two data-flow problems is the mapping function M and the direction: reaching definitions is a forward problem, while live variables is a backward problem.

The desired result of solving data-flow analysis problems is the meet-over-all-paths (MOP) solution:

Definition 3.5.8. *Let $G = (N, E, s)$ be a flow graph with starting point s. For every $b \in N$, let Path(b) be the set of all paths from s to b, and F_b represent the flow function associated with node b. Given a path $p = (p_1, \ldots, p_n) \in Path(b)$, F_p is the composition of the individual F_{p_i}:*

$$F_p = F_{p_1} \circ \ldots \circ F_{p_n}$$

The meet-over-all-paths solution is

$$MOP(b) = \bigsqcup_{p \in Path(b)} F_p(0)$$

for all $b \in N$.

Unfortunately, it is generally undecidable, whether an algorithm exists that computes the MOP solution for all possible flow graphs [Hec77]. The algorithms therefore compute the maximum fixed point (MFP) solution:

Definition 3.5.9. *Let $G = (N, E, s)$ be a flow graph with starting point s. For every $b \in N$, let F_b represent the flow function associated with node b, Pred(b) the set of predecessors, and let the nodes in N be numbered from 1 to n, where n is the number of nodes in N, in reverse postorder. The maximum-fixed-point solution is defined as the maximum fixed point of the following equations:*

$$DF(1) = \bot \qquad DF(i) = \bigsqcup_{p \in Pred(i)} F_b(DF(j)) \qquad 2 \leq i \leq n$$

The MFP solution is a solution of the data-flow analysis equations that is maximal in the ordering of S. For distributive data-flow analysis frameworks, the MFP solution is equal to the MOP solution [Kil73].

The algorithm presented in Figure 3.11 computes the MFP solution for a given instance of a monotone data-flow analysis framework [KU75]. The algorithm maintains a queue of nodes to be processed. Upon startup, this queue is initialized with all nodes in the flow graph. Note that the ordering of the nodes is important: The algorithm achieves maximum performance if the queue contains the nodes in reverse post-order, i.e. a node is visited before any of its successor have been visited. The data-flow information that reaches node $b \in N$ is given by DF(b) and is initialized to the bottom element of the corresponding lattice.

The algorithm consists of a single loop that is iterated as long as the worklist is not empty. In each iteration, the first element of the worklist is removed from the list and the data-flow information for all predecessors is computed and combined with the meet or join operation. If the combined data-flow information from all predecessors of the current node is different from the data-flow information at the node, the node is updated and all successors of the node are appended to the worklist in order to propagate the

Fig. 3.11. Iterative Data-Flow Algorithm

```
void iterate_reach(struct procedure *proc_ptr)
{
list_init(&work_list);

while(bblock_ptr = walk_list(proc_ptr->post_order)) {
 list_append(&work_list, bblock_ptr); }

while(bblock_ptr = walk_list(work_list)) {

 bblock_ptr = list_remove(&work_list, work_list.head_ptr);

 totalmask  =  0;
 while(parent_ptr = walk_list(bblock_ptr->parent_list)) {
  totalmask |= dflow_reach(&work_list, parent_ptr); }

 /* special case for single-bblock procedures */
 if((bblock_ptr->parent_list.num_elements == 0) &&
    (bblock_ptr->childs_list.num_elements == 0)) {
  dflow_reach(&work_list, bblock_ptr);}

 if(totalmask != bblock_ptr->reach_mask1) {
  bblock_ptr1->reach_mask1 = totalmask1;

  if(bblock_ptr->childs_list.num_elements != 0) { /* childless ? */

   /* add successors of current bblock to work_list */
   while(child_ptr = walk_list(bblock_ptr->childs_list)) {
    list_append(&work_list, child_ptr); }
   }
  else { /* special case for childless bblocks */
   dflow_reach(&work_list, bblock_ptr1); }
  }
 else if(bblock_ptr1->childs_list.num_elements == 0) {
  dflow_reach(&work_list, bblock_ptr1); }

 }
}
```

data-flow information. Note that the description above assumes a forward flow problem. The corresponding algorithm for backward flow problems initializes the list in reverse pre-order, computes and combines the data-flow information from all successors and appends all predecessors to the worklist instead.

The runtime of the algorithm depends on the meet or join operation and the complexity of the flow functions. However, the number of loop traversals is bounded by $A + 2$, where A is the maximum number of back edges on any path through the flow graph [HU75]. Note that A can be on the order

of $|N|$, i.e. the number of nodes in the flow graph, but usually $A \leq 3$. While there are more efficient algorithms, the iterative algorithm presented above is simple to implement and widely used.

3.5.7 Register Allocation

Register allocation is a major component of all compilers: Given a set of values that might reside in registers, such as variables, temporaries, and constants, register allocation determines those values that reside in registers. Since the number of registers is usually much smaller than the number of values that might reside in registers, and instructions operate significantly faster on registers than on memory, register allocation is important for performance. This is especially true for RISC processors, where usually all instructions, except load and store instructions, operate on registers only.

Register allocation should not be confused with register assignment, i.e. determining the actual register for the allocated values. Register assignment is a trivial task for modern RISC architectures, as the register sets are usually divided into two uniform sets, the integer and floating-point registers, but are otherwise general purpose.

Early approaches used local methods such as usage counts [Fre74] or bin-packing [Lev83] to solve the register allocation problem. The former approach counts the number of uses and definitions in each basic block for all potential register residing values, e.g. variables, temporaries, constants. These usage counts are used together with the loop depth to prioritize the individual register residing values. Registers are allocated for values in decreasing order of priority.

The latter approach divides all values into groups, ranks all values within each group by priority, and tries several permutations to pack values into registers or memory locations. A similar approach is still used in Digital's GEM compiler system [BCD+92].

Global methods based on graph coloring provide a more effective approach to register allocation. Graph coloring determines the minimum number of colors required to color a graph, such that no two adjacent nodes have the same color. Register allocation can be transformed into a graph coloring problem by mapping values to nodes and connecting two nodes, whenever the corresponding values interfere, i.e. cannot reside in the same register. However, graph coloring is known to be NP-complete [GJ79], hence powerful heuristics are required for an effective algorithm.

The assembler converter uses a register allocator that is based on graph coloring, but is quite different from any of the other register allocators known from literature. As the register allocator is a major component of the assembler converter, it is described in detail: The following sections introduce the graph coloring problem as well as two popular approaches to register allocation based on graph coloring, since the register allocator uses elements from

both approaches. Afterwards, the data-flow analyses used to identify the allocatable objects, i.e. live ranges, as well as the interference model used in the assembler converter is described. Last, the actual allocation algorithm and the algorithm used to determine the location of save and restore instructions are described.

Graph Coloring.

Definition 3.5.10. *Let $G = (V, E)$ be an undirected graph, where V is the set of nodes and E is the set of edges. A k-coloring of such a graph assigns each node one of k different numbers such that all adjacent nodes have different colors. Formally a k-coloring is a function $c : V \to \{1, \ldots, k\}$, such that $c(u) \neq c(v)$ for all edges $(u, v) \in E$. The k-graph-coloring problem is to determine a coloring that uses less than or equal to k colors.*

The k-graph coloring problem is NP-complete for all $k \geq 2$ [GJ79], hence all practical algorithms for computing graph colorings are based on heuristics. Several powerful heuristics exist [BG95], the two approaches to register allocation based on graph coloring presented below use different sets of heuristics.

Register allocation can be transformed into a graph-coloring problem as follows: Determine the set of allocatable objects, and construct a so-called interference graph, where the nodes of the graph represent the individual allocatable objects. Two nodes are connected by an edge if the corresponding objects interfere, i.e. cannot be allocated to the same register since they are in use simultaneously. After the interference graph has been constructed, it is colored with n colors, where n is the number of available registers, i.e. the size of the register set. If the register set is not uniform, a subset of the colors must be associated with every allocatable object. If an n-coloring is found, allocate each object to the register represented by the assigned color.

There are two major approaches to register allocation based on graph coloring: The first approach is called graph coloring and was developed by Chaitin [CAC+81][Cha82]. The second approach is called priority-based graph-coloring and originates with Chow and Hennessy [CH84][CL90]. Both approaches use graph coloring to solve the register allocation problem, but are quite different otherwise, as the following comparison shows.

Chaitin's approach performs register allocation on a machine-level representation, i.e. after code generation, whereas the latter performs register allocation on an intermediate-level representation, i.e. before code generation. Consequently, the former approach uses machine-level instructions as the unit of coloring, while the latter uses basic blocks as the unit of coloring. Chaitin's algorithm can achieve a lower chromatic number, i.e. a coloring that uses fewer colors, due to the smaller units of allocation. Since Chaitin's algorithm operates on the machine-level, all references and definitions of registers are already known, which is not the case for the intermediate representation used in the other approach. Therefore Chaitin's algorithm applies graph coloring to all available registers, whereas the priority-based approach has to reserve several registers, that are used during code generation.

Another major difference between the two approaches is the model of allocation: The priority-based approach uses a pessimistic model, i.e. assigns a memory location for every object and tries to allocate some of these objects to registers. The other approach uses an optimistic model, i.e. assumes that all objects start in registers and generates spill code as necessary. Note that Chaitin's approach always spills objects to the corresponding memory location, while the latter spills objects onto the stack. Due to the different models of allocation, the priority-based approach omits local temporaries, while the other approach omits global temporaries from allocation.

The two approaches use a different model for interferences as well: While Chaitin's algorithm uses live ranges that interfere if one live range is live at the definition point of the other, the priority-based algorithm uses live ranges that interfere if both live ranges are simultaneously live. Since Chaitin's algorithm uses live ranges with finer granularity, i.e. machine-level instructions, than the live ranges of the priority-based algorithm, i.e. basic blocks, the latter approach produces an interference graph that may be less precise in certain situations.

Both approaches use cost-saving estimates to steer the allocation of registers. However, Chaitin's algorithm spills those objects that are least costly to spill, while the priority-based algorithm allocates those objects that benefit the most, another consequence of the different allocation models.

Both approaches use heuristics to lower the chromatic number of the interference graph. Chaitin's algorithm spills live ranges to memory, thereby eliminating the corresponding nodes from the interference graph. However, the spilled object must be restored and saved before and after each reference or definition, respectively, introducing several new nodes to the interference graph. In contrast, the priority-based algorithm splits live ranges into two or more smaller ones, effectively reducing the chromatic number of the interference graph. There is no need to add new nodes to the interference graph for spilled objects, as each object has a corresponding memory location. Note that Chaitin's algorithm either spills an object or allocates a register for it, but never splits live ranges.

The assembler converter uses a register allocator that is based on graph coloring and combines elements from both approaches. The fundamental difference between the register allocator in the assembler converter and traditional register allocators are the allocatable objects: Traditional approaches allocate objects such as variables, temporaries, and constants to registers, while the assembler converter allocates registers to registers.

Spilling of allocatable objects is not possible, as each register must be reallocated to another register. Note that the thread descriptor provides a memory location for each register, hence the allocation model from the priority-based approach is used. As the assembler converter operates on assembler instructions, i.e. after code generation, the register allocator uses

assembler instructions as the unit of allocation, as well as the corresponding interference model.

Since spilling of registers is not an option, the register allocator splits live ranges to lower the chromatic number of the interference graph. Together with the small unit of allocation, this ensures that a coloring without any spills is achieved, as long as live ranges are split early enough. Therefore the register allocation in the assembler converter uses the heuristics from the priority-based graph coloring.

Live Ranges. The register allocator in the assembler converter allocates registers instead of variables, temporaries, and constants like traditional allocators. Therefore some of the definitions need to be translated for this special case:

Definition 3.5.11. *A given point in the control-flow graph is a definition of a register, if the register is written by the corresponding machine instruction. A given point in the control-flow graph is a use of a register, if the register is read by the corresponding machine instruction.*

Definition 3.5.12. *A register is said to be reaching at a particular point in the control-flow graph, if there exists a path in the graph from a definition or use of the register to that particular point. A register is said to be live at a particular point in the control-flow graph, if a path in the graph from that point to an exit node exists that contains a use or definition of the register.*

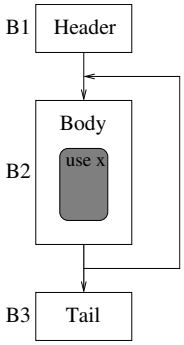
Data-flow analysis can be used to determine the set of nodes in the control-flow graph where a register is live/reaching, respectively. The reaching problem is an instance of the reaching definition problem presented in Section 3.5.6. The only difference is the function M that maps the nodes to a particular flow function: For each node, two m-bit masks k, g are created, where m is the total number of registers. The ith bit of mask g is set if the corresponding instruction contains a use or definition of register number i, cleared otherwise. The ith bit of mask k is only set in the context of external calls. The register masks k, g determine the corresponding flow function $\langle k, g \rangle$.

The same mapping is used to solve the live register problem as an instance of the live variable problem presented in Section 3.5.6. As both problems are distributive data-flow analysis frameworks, the iterative algorithm presented in Section 3.5.6 can be used to compute the meet-over-all-paths solution for the reaching register and live register problems.

Definition 3.5.13. *A live range of a register is a contiguous group of nodes of the control-flow graph, in which a register is reaching and live.*

The current implementation of the assembler converter uses a more restricted form of live ranges that always begin and end at a definition or use of the register. Without this restriction, the efficiency of the allocation may be reduced, as Figure 3.12 illustrates: The register is reaching and live in

Fig. 3.12. Live Range Example



the loop body, thus the insertion of save and restore instructions at the top and bottom of the loop body is required. Note that moving the save/restore instructions out of the loop body may cause the creation of new basic blocks. As the assembler converter does not support optimizations that require a re-layout of the code, this is not possible in the current implementation. However, once the assembler converter is integrated into a compiler, this restriction no longer applies. Using the restricted definition of live ranges, the save and restore instructions are still inside the loop body, but there is an additional available register outside the restricted live range.

Under the restricted definition, a live range is said to start at a given node in the control-flow graph, if the node is a definition or use of the register and satisfies one of the following conditions:

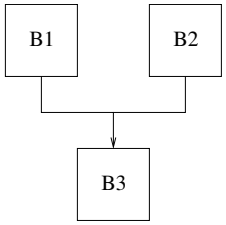
- The register is live and reaching at this node, but not at any of the predecessor nodes.
- Any acyclic path in the control-flow graph from this node to any other node that satisfies the first condition contains neither a use nor a definition of the register.

Similarly, a live range is said to end at a given node in the control-flow graph if the node is a definition or use of the register and satisfies one of the following conditions:

- The register is live and reaching at this node, but not at any of the successor nodes.
- Any acyclic path in the control-flow graph from this node to any other node, that satisfies the first condition, contains neither a use nor a definition of the register.

As the following paragraphs show, the list of nodes that start and end a live range can be determined by data-flow analysis.

Fig. 3.13. Live Range Example



Data-Flow Analysis. Let S be the set of pairs of m-bit vectors, i.e. formally

$$S = \{(n, r) | n, r \text{ are } m\text{-bit vectors}\}$$

where m is the number of registers. Given such a pair (n, r), the elements are interpreted in the following way: The ith bit of the r mask is set if register number i is reaching, cleared otherwise. The ith bit of the n mask is set if an acyclic path in the control-flow graph exists that contains neither a use nor a definition of register number i.

Based on the above interpretation, the meet operation on S is defined as:

$$(n_1, r_1) \sqcup (n_2, r_2) = ((n_1 \vee (n_1 \vee n_2)\overline{r_1}\,\overline{n_1}) \wedge (n_2 \vee (n_1 \vee n_2)\overline{r_2}\,\overline{n_2}), r_1 r_2)$$

Note that the meet operation is identical to the join operation for the reaching definition problem if restricted to the r masks. The operations for the n mask can be interpreted as follows: The ith bit of the n mask is set if the ith bit of both n masks is set, or the ith bit is set in either one of the n masks and the ith bit of the other r mask is cleared, respectively.

The reasoning behind this is illustrated in Figure 3.13. If the ith bit of both n masks is set, there exists a path from any of the two predecessor nodes to any node that satisfies the first condition, i.e. register i is live and reaching at that node, but not at any of the preceding nodes. Consequently, no such paths exists that starts at node B2. If the ith bit of the n masks at either node B1 or B2 is set, the ith bit of the reaching mask is cleared at node B2 or B1, respectively. Since register i is neither reaching at nodes B1 or B2, a path that satisfies the second condition cannot exist.

Lemma 3.5.8. *The tuple $(S, \sqcup)$ as defined above forms a semi-lattice.*

Proof. Note that the proofs for the reach-masks are omitted, as the meet operation on these masks is identical to the join operation defined for the reaching definition problem, which was proven to be a semi-lattice in Section 3.5.6.

The $\sqcup$ operation is idempotent:

$$\begin{aligned}(n_1, r_1) \sqcup (n_1, r_1) &= (n_1 \vee (n_1 \vee n_1)\overline{r_1}\,\overline{n_1}) \wedge (n_1 \vee (n_1 \vee n_1)\overline{r_1}\,\overline{n_1}) \\ &= (n_1 \vee (n_1 \vee n_1)\overline{r_1}\,\overline{n_1}) \\ &= (n_1 \vee n_1\overline{r_1}\,\overline{n_1}) \\ &= n_1\end{aligned}$$

The $\sqcup$ operation is reflexive:

$$\begin{aligned}(n_1, r_1) \sqcup (n_2, r_2) &= (n_1 \vee (n_1 \vee n_2)\overline{r_1}\,\overline{n_1}) \wedge (n_2 \vee (n_1 \vee n_2)\overline{r_2}\,\overline{n_2}) \\ &= (n_2 \vee (n_2 \vee n_1)\overline{r_2}\,\overline{n_2}) \wedge (n_1 \vee (n_2 \vee n_1)\overline{r_1}\,\overline{n_1}) \\ &= (n_2, r_2) \sqcup (n_1, r_1)\end{aligned}$$

The $\sqcup$ operation is associative:

$$\begin{aligned}((n_1, r_1) \sqcup (n_2, r_2)) \sqcup (n_3, r_3) &= (n_1 \vee (n_1 \vee n_2)\overline{r_1}\,\overline{n_1}) \wedge \\ &\quad (n_2 \vee (n_1 \vee n_2)\overline{r_2}\,\overline{n_2}) \sqcup (n_3, r_3) \\ &= \underbrace{(n_1n_2 \vee n_1\overline{r_2}\,\overline{n_2} \vee n_2\overline{r_1}\,\overline{n_1})}_{n'} \sqcup (n_3, r_3) \\ &= n'n_3 \vee n'\overline{r_3}\,\overline{n_3} \vee n_3\overline{r'}\,\overline{n'} \\ &= (n_1n_2n_3 \vee n_1\overline{n_2}n_3\overline{r_2} \vee \overline{n_1}n_2n_3\overline{r_1}) \vee \\ &\quad (n_1n_2\overline{n_3}\,\overline{r_3} \vee n_1\overline{n_2}\,\overline{n_3}\,\overline{r_3} \vee \overline{n_1}n_2\overline{n_3}\,\overline{r_1}\,\overline{r_3}) \vee \\ &\quad \overline{(n_1n_2 \vee n_1\overline{r_2}\,\overline{n_2} \vee n_2\overline{r_1}\,\overline{n_1})}\,\overline{r_1}\,\overline{r_2}n_3 \\ &= (n_2n_3n_1 \vee n_2\overline{n_3}n_1\overline{r_3} \vee \overline{n_2}n_3n_1\overline{r_2}) \vee \\ &\quad (n_2n_3\overline{n_1}\,\overline{r_1} \vee n_2\overline{n_3}\,\overline{n_1}\,\overline{r_1} \vee \overline{n_2}n_3\overline{n_1}\,\overline{r_2}\,\overline{r_1}) \vee \\ &\quad \overline{(n_2n_3 \vee n_2\overline{r_3}\,\overline{n_3} \vee n_3\overline{r_2}\,\overline{n_2})}\,\overline{r_2}\,\overline{r_3}n_1 \\ &= (n_1, r_1) \sqcup ((n_2 \vee (n_2 \vee n_3)\overline{r_2}\,\overline{n_2}) \wedge \\ &\quad (n_3 \vee (n_2 \vee n_3)\overline{r_3}\,\overline{n_3})) \\ &= (n_1, r_1) \sqcup ((n_2, r_2) \sqcup (n_3, r_3))\end{aligned}$$

Let the set of operations F be defined as follows:

$$F = \{\langle k_1, g_1, k_2, g_2\rangle(x) \mid \langle k_1, g_1, k_2, g_2\rangle(x_1, x_2) = ((x_1 \wedge \overline{k_1}) \vee g_1, (x_2 \wedge \overline{k_2}) \vee g_2)\}$$

The mapping M of nodes in the control-flow graph to operations in F is determined as follows: The ith bit of the k_1, g_2 masks is set if the node contains a use/definition of register number i, cleared otherwise. The ith bit of k_2 is set in connection with external calls, cleared otherwise. The ith bit of g_1 is set if the node is a loop header and register number i is reaching and live inside the loop body, but not outside.

Lemma 3.5.9. *The triple $(S, \sqcup, F)$ is a distributive data-flow analysis framework.*

Proof. The monotonicity is implied by the distributiveness shown below, the identity operation in F is given by $\langle 0,0,0,0\rangle$:

$$\langle 0,0,0,0\rangle(x) = ((x \wedge \overline{0}) \vee 0) = x \qquad \forall x \in F$$

Let $\langle k_1^1, g_1^1, k_2^1, g_2^1\rangle$, $\langle k_1^2, g_1^2, k_2^2, g_2^2\rangle$ be elements of F. The composition of these elements is an element in F:

$$\begin{aligned}\langle k_1^1, g_1^1, k_2^1, g_2^1\rangle(\langle k_1^2, g_1^2, k_2^2, g_2^2\rangle(x)) &= \langle k_1^1, g_1^1, k_2^1, g_2^1\rangle((x \wedge \overline{k_2^2}) \vee g_2^2) \\ &= (((x \wedge \overline{k_2^2}) \vee g_2^2) \wedge \overline{k_2^1}) \vee g_2^2 \\ &= ((x \wedge (\overline{k_2^2} \wedge \overline{k_2^1})) \vee (g_2^2 \wedge \overline{k_2^1} \vee g_2^1) \\ &= \langle (\overline{k_2^2} \wedge \overline{k_2^1}), (g_2^2 \wedge \overline{k_2^1} \vee g_2^1)\rangle\end{aligned}$$

For $x \in S$, the function $\langle 0, x, 0, x\rangle$ meets the condition:

$$\langle 0, x, 0, x\rangle(0) = (0 \wedge \overline{0}) \vee x = x$$

The $\sqcup$ operation is distributive, there are two different cases:

- Suppose that the ith bit of $x \sqcup y$ is zero, i.e.

$$(n_1 \vee (n_1 \vee n_2)\overline{r_1}\,\overline{n_1}) \wedge \ (n_2 \vee (n_1 \vee n_2)\overline{r_2}\,\overline{n_2}) = 0$$

This situation arises in the following three cases:

$$n_1 = 0, n_2 = 0 \text{ and } n_1 = 0, n_2 = 1, r_1 = 1 \text{ and } n_1 = 1, n_2 = 0, r_2 = 1$$

The last two cases are symmetric, hence only the first two cases are shown.

– The ith bit of $f(x), f(y)$ equals the ith bit of g_1, since

$$((0 \wedge \overline{k_1}) \vee g_1) = g_1$$

As the $\sqcup$ operator is idempotent, $f(x) \sqcup f(y)$ equals the ith bit of g_1, which equals the ith bit of $f(x \sqcup y)$. Note that the ith bit of $x \sqcup y$ was supposed to be zero.

– The ith bit of $f(x)$ equals the ith bit of g using the same reasoning as above. The ith bit of $f(y)$ equals the ith bit of

$$((1 \wedge \overline{k_1}) \vee g_1) = (\overline{k_1} \vee g_1)$$

Therefore the ith bit of $f(x) \sqcup f(y)$ equals

$$\begin{aligned}&= (g_i \vee (g_1 \vee (\overline{k_1} \vee g_1))\overline{r_1}\,\overline{g_1}) \wedge ((\overline{k_1} \vee g_1) \vee (g_1 \vee (\overline{k_1} \vee g_1))\overline{r_2}(\overline{\overline{k_1} \vee g_1})) \\ &= (g_1((\overline{k_1} \vee g_1) \vee (\overline{k_1} \vee g_1)\overline{r_2}(\overline{g_i}\,\overline{\overline{k_1}}))) \\ &= (g_1(g_1 \vee \overline{k_1}) \\ &= g_1\end{aligned}$$

which equals the ith bit of $f(x \sqcup y)$ using the same reasoning as in the first case.

- Assume that $x \sqcup y = 1$, i.e.

$$(n_1 \vee (n_1 \vee n_2)\overline{r_1}\,\overline{n_1}) \wedge (n_2 \vee (n_1 \vee n_2)\overline{r_2}\,\overline{n_2}) = 1$$

This situation arises in the following three cases:

$$n_1 = 1, n_2 = 1 \text{ and } n_1 = 0, n_2 = 1, r_1 = 0 \text{ and } n_1 = 1, n_2 = 0, r_2 = 0$$

The last two cases are symmetric, hence only the first two cases are shown.

– The ith bit of $f(x), f(y)$ equals the ith bit of $(\overline{k_1} \vee g_1)$ as

$$(1 \wedge \overline{k_1}) \vee g_1 = \overline{k_i} \vee g_1$$

As the $\sqcup$ operator is idempotent, the ith bit of $f(x) \sqcup f(y)$ equals the ith bit of $\overline{k_1} \vee g_1$ as well. Since the ith bit of $x \sqcup y$ is supposed to be one, the ith bit of $f(x \sqcup y)$ equals the ith bit of $\overline{k_1} \vee g_1$ using the same reasoning.

– The ith bit of $f(x)$ equals the ith bit of g_1, since

$$(0 \wedge \overline{k_1}) \vee g_1 = g_1$$

while the ith bit of $f(y)$equals the ith bit of $(\overline{k_1} \vee g_1)$ as

$$(1 \wedge \overline{k_1}) \vee g_1 = \overline{k_1} \vee g_1$$

The ith bit of $f(x) \sqcup f(y)$ is

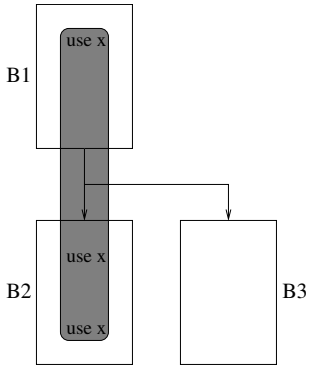
$$\begin{aligned}
&= \left((\overline{k_i} \vee g_i) \vee (g_i \vee (\overline{k_i} \vee g_i))\overline{r_2}\overline{(\overline{k_i} \vee g_i)}\right) \wedge (g_i \vee (g_i \vee (\overline{k_i} \vee g_i))\overline{r_1}\,\overline{g_i}) \\
&= ((\overline{k_i} \vee g_i) \vee (\overline{k_i} \vee g_i)\overline{r_2}\overline{(\overline{k_i} \vee g_i)} \wedge (g_i \vee (\overline{k_i})\overline{g_i} \vee g_i) \\
&= (\overline{k_i} \vee g_i)
\end{aligned}$$

Since the ith bit of $x \sqcup y$ is supposed to be one, the ith bit of $f(x \sqcup y)$ equals the ith bit of $\overline{k_i} \vee g_i$.

The end points of a live range in the control-flow graph can be determined in the same way: The only difference is the direction of the data-flow analysis and the mapping M, that maps a node to the corresponding flow function: The ith bit of the k_1, g_2 masks is set if the node contains a use or definition of register number i, cleared otherwise. The ith bit of k_2 is set in connection with external calls, cleared otherwise. The ith bit of g_1 is set if the node is a loop header and register number i is reaching and live inside loop body, but not outside.

After the four data-flow analyses presented above have been completed, all nodes in the control-flow graph, that are the start or end of a live range, are marked accordingly. Note that live ranges can only start at nodes, but can end at nodes or edges, as Figure 3.14 shows. Therefore all edges are inspected and those that are an end point for a live range are marked as well. Afterwards the actual live range data structures are created.

Fig. 3.14. Live Range Example



Splitting & Merging. The live range data structure contains the following elements:

- A mask of registers that are available for this live range.
- A mask of registers that are forbidden for this live range.
- A priority that is used to steer the allocation of live ranges.
- The number of instructions in the live range as well as the number of use and definitions of the corresponding register in the live range.
- A list of interfering live ranges, i.e. live ranges that cannot be allocated to the same register as the live range itself.
- A list of subranges, one for each starting point of the live range.

Each subrange contains those instructions in depth-first search order from the corresponding live range, that are reachable from the starting point of the subrange and are not in any of the other subranges. As the live range is a contiguous set of nodes, the individual subranges connect. The instructions in a subrange form a tree with the corresponding starting point as the root, while the instructions on the whole live range form a forest with one tree for each starting point in the live range.

The data structures are created by allocating a new data structure at each starting point of a live range and adding instructions to the live range until an endpoint is encountered on all paths from the starting point. Note that all live ranges created in this way contain only one subrange, hence live ranges overlap if a live range has more than one starting point.

The individual live ranges are now splitted at super block boundaries to ensure that no live range extends across a context switch. Note that this splitting of live ranges is omitted if register partitioning is used. The splitting algorithm exploits the fact that all live ranges contain only one subrange at this point and works in the following way:

The algorithm processes all basic blocks and checks them for the end-of-sblock flag. If the current basic block ends an super block, all children of the block are processed. For each child block, the list of live ranges in the current basic block is compared with the list of live ranges in the child. If a match is found, i.e. a live range that crosses the super block boundary, the live range is split across the corresponding edge.

The split is performed in two steps: First, all instructions in the live range that can be reached from the top of the child block are extracted from the original live range and moved to a new live range. Note that the remaining instructions in the original live range are still in depth-first search order, but the live range may no longer end at a use/definition point of the corresponding register. Therefore all loose ends are removed from the list of instructions in the original live range in order to maintain the live range properties.

The extracted instructions are used to spawn one or more live ranges. The spawning of live ranges is a special case of the general algorithm described below, therefore a description of the algorithm is omitted here. The only difference between the two algorithms is that the one used during initial splitting of live ranges operates only in one direction, as each live range contains only one subrange, and ignores interferences between live ranges, as the interference graph has not been constructed yet.

The splitting pass ensures that all live ranges are contained in super blocks, at least in the absence of register partitioning. But the live ranges still consist of a single subrange and several subranges may overlap. Hence the overlapping live ranges are merged by the algorithm described in Figure 3.15, taken from [Muc97].

For each register, the algorithm separates all live ranges for that register from the total list of live ranges and stores these live ranges in a temporary list. This separation significantly reduces the number of live ranges that have to be processed in each step, thereby increasing performance. Note that only live ranges for the same registers can be merged.

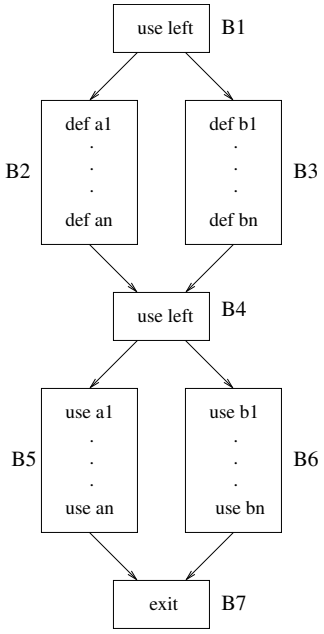
The algorithm uses several rounds to merge all live ranges for the current register, initiating a new round as long as the stop flag is cleared. The stop flag is set at the beginning of each round, but is cleared if at least two live ranges were merged, since the merged live range may now overlap with other live ranges. In each round, all pairs of live ranges are processed. If both live ranges in the pair share a use or definition, i.e. overlap, these live ranges are merged. The algorithm does always converge since the number of live ranges is finite and decreases with every merge operation.

This condition is checked by processing all instructions in one live range and checking for each use and definition point whether it is present in the other live range. The merging is performed by moving all subranges from one live range to the other live range, such that the individual subranges in the merged live range do not overlap. The remaining empty live range is removed from the temporary list and destroyed afterwards.

Fig. 3.15. Merging of Live Ranges

```
void merge_graph(struct IFgraph *graph_ptr)
{

list_init(&LRlist);

for(i = 0;i < REG_NUM_REGS;i++) {

 while(lrange_ptr1 = walk_list(&graph_ptr->constrained)) {
  if(lrange_ptr1->orig_reg->pnum == i) {
   list_append(&LRlist, lrange_ptr1); }
 }

stop = 0;
while(!stop) {
 stop = 1;

 while(lrange_ptr1 = walk_list(LRlist)) {

  lrange_ptr2 = lrange_ptr1->next_ptr;
  while(lrange_ptr2) {

    while(srange_ptr = walk_list(lrange_ptr1->SRlist) && stop) {

     while(inst_ptr = walk_list(srange_ptr->inst_list) && stop) {

      if((inst_ptr->flag & ALC_PD_DEF) ||
         (inst_ptr->flag & ALC_PD_USE)) {
       if(search_PDitem1(lrange_ptr2, inst_ptr)) {
         merge_lranges(lrange_ptr1, lrange_ptr2); stop = 0; }
       }
      }
     }

   lrange_ptr2 = lrange_ptr2->next_ptr;
   }
  }
 }

 while(lrange_ptr1 = list_remove(&LRlist, LRlist.head_ptr)) {
  list_append(&graph_ptr->constrained, lrange_ptr1); }
 }
}
```

Fig. 3.16. Interference Example



After all overlapping live ranges for the current register have been merged, the live ranges in the temporary list are moved to the total list of live ranges. Once all live ranges for all registers have been merged, i.e. the final shape of all live ranges is known, the interference graph is constructed as described below.

Interferences. The two register allocators described above use different models for the interferences between live ranges: In Chaitin's algorithm, two live ranges interfere, if they overlap at a definition point of one or both live ranges. A definition point is a node in the control-flow graph that contains a use or definition of the corresponding register. In the priority-based algorithm, two live ranges interfere if they overlap at an arbitrary point, hence providing a less precise model of interference.

The difference between these two models is illustrated by looking at the example in Figure 3.16 taken from [Muc97]. In the interference model used by the priority-based algorithm, there are interferences between the $a_1, \ldots, a_n$, $b_1, \ldots, b_n$, and left, i.e. the interference graph has $2n+1$ nodes and $2n(2n+1)$ edges. As block B_4 is neither a definition point for the $a_1, \ldots, a_n$, nor for the $b_1, \ldots, b_n$, the a_i do not interfere with the b_i at all if the interference model from Chaitin's algorithm is used. Note that the corresponding interference graph has $2n + 1$ nodes and $n(n + 1)$ edges.

Fig. 3.17. Interference Graph Construction

```
void build_graph(struct procedure *proc_ptr,
                 struct IFgraph *graph_ptr)
{

while(bblock_ptr = walk_list(&proc_ptr->pre_order)) {

 while(inst_ptr = walk_list(&bblock_ptr->inst_list) {

  if(inst_ptr->flag & INST_INSTRUCTION)
   while(lrange_ptr1 = walk_list(&inst_ptr->LRlist)) {
    reg_mask1 = pnum2mask(lrange_ptr1->orig_reg->pnum);

    while(lrange_ptr2 = walk_list(&inst_ptr->LRlist)) {
     reg_mask2 = pnum2mask(lrange_ptr2->orig_reg->pnum);

     if(((inst_ptr->def_mask | inst_ptr->use_mask) & reg_mask1) ||
        ((inst_ptr->def_mask | inst_ptr->use_mask) & reg_mask2))

      if((reg_mask1 & reg_mask2) == 0)
       if(!(((LRitem_ptr1->flag & ALC_LR_END) &&
             (reg_mask1 & inst_ptr->use_mask) &&
             (LRitem_ptr2->flag & ALC_LR_NEW) &&
             (reg_mask2 & inst_ptr->def_mask) &&
            !(reg_mask2 & inst_ptr->use_mask)) ||
            ((LRitem_ptr2->flag & ALC_LR_END) &&
             (reg_mask2 & inst_ptr->use_mask) &&
             (LRitem_ptr1->flag & ALC_LR_NEW) &&
             (reg_mask1 & inst_ptr->def_mask) &&
            !(reg_mask1 & inst_ptr->use_mask)))) {
        link_lranges(LRitem_ptr1->lrange, LRitem_ptr2->lrange); }
       else if(inst_ptr->use_mask & reg_mask1)
        if(inst_ptr->rst_after & reg_mask1) {
         inst_ptr->rst_after  &= ~reg_mask1;
         inst_ptr->rst_before |=  reg_mask1; }
       else if(inst_ptr->use_mask & reg_mask2)
        if(inst_ptr->rst_after & reg_mask2) {
         inst_ptr->rst_after  &= ~reg_mask2;
         inst_ptr->rst_before |=  reg_mask2; }
     }
    }
}
```

Given a list of live ranges, the algorithm in Figure 3.17 creates the corresponding interference graph. The algorithm processes all instructions in the procedure in basic block order. For each instruction, each pair of live ranges in the list of live ranges at the instruction is inspected. If the current instruction is a definition point for at least one of the two live ranges, the corresponding interferences are added to both live ranges.

However, if one of the live ranges ends in a use, while the other starts with a definition, both live ranges do not interfere: The save instruction for the live range that ends can be inserted before the current instruction, no restore instruction is necessary for the live range that starts at the current instruction.

Coloring. The register allocator uses priority-based graph coloring to color the interference graph constructed above. The interference graph consists of three lists: the list of constrained, unconstrained, and finished live ranges. A live range is constrained if the number of interferences is larger than or equal than the number of available registers, unconstrained otherwise. A live range is finished if it has already been colored.

The interferences themselves are stored at each live range, i.e. each live range contains a list of interfering live ranges. Note that after construction of the interference graph, all live ranges reside in the constrained list. The coloring algorithm works as described in the following paragraphs:

All live ranges that have only one available register, i.e. are used in an external call prologue or epilogue, are colored. These live ranges are identified by processing all live ranges in the list of constrained live ranges. After coloring these live ranges, the interfering live ranges are updated by marking the allocated register as forbidden. The colored live range is then moved from the constrained list to the finished list.

All unconstrained live ranges are identified and moved to the unconstrained list. The live ranges in the unconstrained list can be easily colored after all constrained live ranges have been colored, as the number of available registers is larger than the number of interferences for these live ranges. Unconstrained live ranges can become constrained and vice versa due to the splitting of live ranges described below.

The following steps are repeated until all constrained live ranges have been colored:

- Compute the priority function p from the priority-based algorithm as a measure of the relative benefits of assigning a register to a given live range:

$$p = \frac{\#\ \text{uses} * \text{LD\_SAVE} + \#\ \text{defs} * \text{ST\_SAVE}}{\#\ \text{instructions}}$$

  The priority function uses two machine specific parameters: LD_SAVE is the cost associated with saving a register to the thread descriptor, ST_SAVE is the cost associated with restoring a register from the thread descriptor. Note that the priority function is normalized by the number of

instructions in the live range to favor smaller live ranges that provide the same benefit.
- The live range with the highest priority is colored. The coloring algorithm uses bit vector operations on the masks of the available and forbidden registers in the live range to determine the set of registers that is available and not forbidden. If the ropt optimization is enabled, one of the registers from the set is selected at random, otherwise the register with the smallest number is selected.
- After coloring, the live range is removed from the constrained list and appended to the list of finished live ranges.
- All live ranges that interfere with the live range that was colored in the previous step, are updated by marking the allocated register as forbidden. In addition, each live range is checked, whether it should be split in order to lower the chromatic number of the interference graph. A live range is split if either all available registers are forbidden, or the number of interfering live ranges is RT_SPLIT times larger than the number of remaining available registers. This heuristic is taken from the priority-based approach and ensures that splitting is initiated early enough to avoid deadlock situations. The RT_SPLIT parameter is set to two as suggested by Chow/Hennessy [CL90].

After all constrained live ranges have been colored, the unconstrained live ranges are colored. As the number of available registers is larger than the number of interferences and therefore the number of forbidden registers, these live ranges can be colored without further splitting of live ranges. After coloring an unconstrained live range, the interfering live ranges are updated by marking the allocated register as forbidden.

After the interference graph has been colored, all colored live ranges reside in the list of finished live ranges. The results of the coloring are applied to the instructions of the current procedure in the following way: For each live range, all instructions that belong to the live range and contain a use/definition of the corresponding register, are updated with the newly allocated register.

Splitting. The splitting process works by spawning one or more live ranges from the top of each subrange in the original live range. Due to the smaller unit of allocation, i.e. assembler instructions, the splitting process is more complex than the one proposed for priority-based graph coloring.

The splitting algorithm operates on regions of the original live range: A region is a continuous part of the live range that contains no more than one use or definition of the corresponding register and is bounded by the start or end of a basic block. These regions are maintained in the boundary list as well as two candidate lists. The boundary list contains regions that cannot be added to the spawned live range. The elements of this list are used to spawn other live ranges. The candidate lists contain regions that can still be added to the spawned live range, sorted in decreasing order of priority.

The corresponding priority function is similar to the one used in the coloring algorithm.

Given an instruction that belongs to the original live range, the algorithm spawns new live ranges in the following way:

Starting with the given instruction, instructions are examined and removed from the original live range in program-flow order until either a basic block boundary or a use or definition of the corresponding register is found. If a basic block boundary is encountered, regions are created for every child block that belongs to the live range and appended to the boundary list. The algorithm is then recursively called for each item in the boundary list.

If a use or definition of the corresponding register is found, a region is created and appended to the candidate list. This ensures that the new live range starts at a use or definition of the corresponding register. The algorithm then repeats the following steps until the candidate list is empty:

- The first region of the candidate list, i.e. the one with the highest priority, is selected and subsequently removed from the list. The instructions that belong to this region are moved to the spawned live range in program-flow or reverse program-flow order, as indicated by the type of the region. The region contains a pointer to the anchor point, i.e. the instruction where the spawned live range and the region connect. Note that the anchor point must already belong to the spawned live range. Since the live range properties must be maintained at all times, moving the instructions may cause the addition of subranges to the spawned live range.
- Since the spawned live range was changed during addition of the region in the previous step, all candidate regions must be reexamined in order to determine whether they can still be added to the spawned live range.
  Therefore each region is processed in the following way: If the region overlaps completely with the region that was added in the previous step, and both regions use the same direction, the region is discarded. If both regions overlap completely, but use different directions, the subrange associated with the region in reverse program-flow order is folded into the subrange that is associated with the other region. The join operation is necessary to maintain the live range properties, e.g. the ordering of the instructions. If both regions overlap at their boundaries, both regions are marked.
  If the current region is not overlapping completely, the size and the priority of the region are recalculated. The priority p of a region is given by the number of additional interferences that would be caused by adding the region, normalized by the number of instructions in the region, i.e.

$$p = 1.0 - \frac{\#\text{ interferences}}{\#\text{ instructions}}$$

  If the region can still be added to the spawned live range, it is moved to the second candidate list. Otherwise, the region is moved to the boundary list.

- After all regions in the candidate list have been evaluated, all regions in the first candidate list have either been moved to the second candidate list or the boundary list. The second candidate list contains all regions that can still be added to the spawned live range. Hence the two candidate lists are exchanged. Afterwards new candidates that connect at the recently added region are explored as described below.
  If the current region was added in program-flow order, new regions are explored in three different ways: If the first instruction of the region is at the start of a basic block, all parents of the corresponding block that belong to the original live range are explored. If the last instruction of the region is at the end of a basic block, all children of the corresponding block that belong to the original live range are explored. If the last instruction of the current region is not at the end of an basic block, a new region is explored starting from the next instruction in the corresponding block. In all three cases, the explored regions are added to the candidate list if they can be added to the spawned live range, else the regions are added to the boundary list.
  If the region was added in reverse program-flow order, new regions are explored in three different ways: If the first instruction of the region is at the end of a basic block, all children of the corresponding block that belong to the original live range are explored. If the last instruction of the region is at the start of a basic block, all parents of the corresponding block, that belong to the original live range are explored. If the last instruction is not at the start of a basic block, a new region is explored starting from the previous instruction in the basic block. In all three cases the explored regions are added to the candidate list if they can be added to the spawned live range. Otherwise the individual regions are added to the boundary list.

After all regions in the candidate list have been added to the spawned live range, the spawned live range may violate live range properties, e.g. may not start or end at a use or definition of the corresponding register. In order to maintain live range properties, any dangling heads and tails are removed from the spawned live range. After the final shape of the spawned live range has been determined, the save and restore flags are calculated as described in Section 3.5.7.

Since the shape of the original live range was altered and a new live range is added, the interference graph has to be updated. Note that the modifications are restricted to live ranges that interfere with the original live range. Therefore the interference graph is updated by processing all live ranges that interfere with the original live range. There are four different cases:

- The live range interferes neither with the original nor with the new live range. Therefore the interference to the original live range can be removed. Since the number of interferences decreases, the live range may become

unconstrained. In that case, the live range is moved from the list of constrained live ranges to the list of unconstrained live ranges.
- The live range still interferes with the original live range, but not with the spawned live range, i.e. no changes are made to the interference graph.
- The live range interferes with the spawned live range, but no longer interferes with the original live range. Therefore the interference with the original live range is replaced by the interference with the spawned live range. Note that the number of interferences is unchanged for the interfering live range, but decreases for the original live range. However, the original live range is deleted after the spawning process, hence there is no need to check whether the original live range is still constrained.
- The live range interferes with the original as well as the spawned live range. The live range already interfered with the original live range, hence it is sufficient to add the interference to the spawned live range. As this increases the number of interferences for the live range, the live range may become constrained. In that case, the live range is moved from the list of unconstrained live ranges to the list of constrained live ranges.

After all interferences have been updated and the spawned live range is non-empty, it is appended to the list of constrained or unconstrained live ranges depending on the number of interferences. If the live range is empty, i.e. contains no instructions, it is discarded.

Finally, all regions in the boundary list are used to recursively spawn new live ranges. This ensures that all uses/definitions in the original live range are covered by the spawned live ranges.

Context Switch Code. Save and restore instructions that save and restore the allocated registers to and from the corresponding locations in the thread descriptor must be inserted at the boundaries of a live range, respectively. Therefore every instruction contains four additional register masks, i.e. RLoadBefore, RLoadAfter, RStoreBefore, RStoreAfter.

If the ith bit of the RLoadBefore mask is set, a restore operation for register number i is inserted before the current instruction. If the ith bit of the RLoadAfter mask is set, a restore operation for register number i is inserted after the current instruction. If the ith bit of the RStoreBefore mask is set, a save operation for register number i is inserted before the current instruction. If the ith bit of the RStoreAfter mask is set, a save operation for register number i is inserted after the current instruction.

The individual masks are updated during the creation of live ranges by processing all instructions of the live range. Recall that live ranges are either created during initial live range creation or by splitting of live ranges during register allocation. The following rules govern the insertion of save and restore instructions:

- Save and restore instructions are only inserted at use or definition points of the corresponding register.

- A restore instruction before a use is required if a path from the use to an instruction outside the live range exists that does not contain any other use or definitions.
- A save instruction after a use or definition is required if the use or definition is at the end of the live range or a path from the use or definition to a restore instruction exists, that is inside the live range.
- A save instruction before a use is used instead of a save instruction after a use, if the instruction is a branch at the end of a super block, or the live range overlaps, but does not interfere, with another live range at the current instruction.

Note that the RLoadAfter mask is currently unused, since the placement of save and restore instruction is not optimized by moving such instructions outside of loops. However, this optimization may cause the creation of additional basic blocks, thereby requiring a re-layout of the code, which is not supported in the current implementation.

Algorithm. The placement of save and restore instructions can be solved by data-flow analysis for each live range. The placement of the restore instructions is a special case ($m = 1$) of the distributive data-flow analysis framework for reaching definitions defined in Section 3.5.6. The single bit is interpreted as follows: The bit is set if a path from an instruction outside of the live range to the current instruction exists that contains no use or definition of the corresponding register, cleared otherwise.

Note that a live range constitutes a flow graph, although it may have multiple entry points. Instead of operating on basic blocks, the data-flow analysis operates on regions that are bounded by live range or basic block boundaries. These two facts require slight modifications to the iterative data-flow algorithm presented in Section 3.5.6.

The mapping of regions to functions from the operation space is done as follows: If the start of the region is on a live range boundary, i.e. one of the predecessor instructions is outside the live range, the g bit is set. If the region contains a use or definition of the corresponding register, the k bit is set and the g bit is cleared if it was set by the previous rule. Note that a restore instruction is only inserted if the incoming bit is set and the first occurrence is a use.

As described above, the placement of restore instructions can be transformed into an instance of the reaching definition problem with $m = 1$. The placement of save instructions can be transformed into an instance of the live variable data-flow problem with $m = 1$ in the same way. The only difference is the interpretation of the data-flow information as well as the mapping of regions to functions from the operation space.

The g bit is set if a path from an instruction outside of the live range to the current instruction exists that contains no use or definition of the corresponding register, cleared otherwise. If the end point of a region is on a live range boundary, i.e. one of the successor instructions is outside the

live range, the g bit is set. If the region contains a use or definition of the corresponding register, the k bit is set and the g bit is cleared if set by the previous rule. A save instruction is only inserted before or after the last use or definition in the region, if the individual data-flow bit is set. However, if the last use or definition is inside a call prologue and the corresponding register is an argument register of the call, the save instruction is omitted, since the content of the register is destroyed by the call.

The individual masks are recalculated for every new live range, either during initial live range creation or during splitting of live ranges. After the interference graph has been colored, the individual register masks are updated in a final pass over all live ranges. This pass identifies all live ranges that overlap, but do not interfere, at a definition point. Recall from Section 3.5.7 that this situation occurs if one live range ends with a use at that point, while the other starts with a definition.

If two such live ranges are encountered, the masks are modified in the following way: If the RStoreAfter mask contains an entry for the register corresponding to the live range that ends with a use, the entry is cleared and set in the RStoreBefore mask instead. This moves the corresponding restore instruction before the current instruction. Since the last access of the associated register is a use, this does not change program semantics.

The algorithm used to determine such pairs of live ranges operates in the same way as the algorithm used to construct the interference graph.

Optimization. If the rstore optimization is enabled, an optional pass over the procedure flow graph removes any restore instructions that are not needed. A restore instruction can be removed if it is the last restore instruction on a path to an exit node and the corresponding register is not used by the calling procedure. This applies to all registers with the exception of return value and callee-save registers. The following algorithm is used to identify such restore instructions:

- All exit basic blocks of the procedure, i.e. basic blocks with no children, are identified and put into a worklist. For all registers that are neither return value nor callee-save registers, the corresponding bit is set in the ExitMask associated with each exit block, cleared otherwise.
- The algorithm processes basic blocks until the worklist is empty. The first basic block in the worklist is removed and the individual instructions in the block are processed to update the ExitMask. If the ith bit of the mask is set and the block contains a restore instruction for the corresponding register, the bit is cleared and the restore instruction is removed. If the ith bit of the exit mask is set and the first occurrence is a use, the bit is cleared, but the restore instruction is left in place.
  If the block contains a call prolog, the corresponding bits in the ExitMask for all argument registers are cleared. Otherwise the restores prior to the call would be removed, causing the subsequent loads in the call prologue to restore outdated register contents.

- After the bit mask of the current block is updated, the information is propagated to the parent blocks in the following way: Each parent is appended to the worklist after all children of the parent have been processed and the logical and of the ExitMasks from all children is non-zero. The ExitMask of the parent is set to the logical bit-vector and of all child masks.

The algorithm presented above ensures that all unnecessary save instructions are removed on paths to exit nodes. However, the performance impact of this optimization is usually negligible, hence this optimization is not enabled by default.

3.5.8 Code Conversion

The actual code conversion process is quite simple, the main tasks were already performed during register allocation. The conversion is performed in three steps: In the first step, the instruction sequence is updated by inserting the required instructions, e.g. save and restore operations to the thread descriptor. In the second step, the instructions are modified based on the informations from the platform-specific configuration file. Last but not least, the modified instructions are printed to the output file. The following paragraphs describe the steps in detail.

Updating Instructions. The instructions are updated separately for each super block in several steps: In the first and last step, the level information is cleared for all basic blocks that belong to the super block. This is required, since the basic blocks are updated in depth-first order and the level information is used to mark those basic blocks that have already been updated. Before the individual basic blocks are updated, all special registers that are used in this super block, e.g. the floating-point control register, have to be restored. Therefore instructions that restore the contents of these special registers from the thread descriptor are inserted at the top of the header block.

Afterwards the individual basic blocks are updated by traversing the blocks via depth-first search, starting with the header block. Note that the subgraph formed by the basic blocks of a super block is an abstract flow graph of the control-flow graph, i.e. all basic blocks are reachable from the header block.

Upon entry to the basic block, the level information is checked. If the level is non-zero, the block has already been updated and the routine returns immediately. If the level is zero, the level is incremented prior to updating all instructions in the basic block in sequential order. After the instructions have been updated, the children of the current block are processed recursively as long as the basic block does not end the current super block.

For each instruction, the necessary save and restore instruction are inserted before and after the instruction in the following way: The live ranges associated with the instruction are examined and the corresponding register mask is determined. This register mask is compared with the RLoadBefore,

RLoadAfter, RStoreBefore and RStoreAfter fields of the instruction that were set up during register allocation. If a match is found between the live mask and the RStoreBefore, RLoadBefore masks, the corresponding save or restore instructions are inserted before the current instruction, respectively. If a match is found between the live mask and the RStoreAfter, RLoadAfter masks, the corresponding save or restore instructions are inserted after the current instruction, respectively.

If the current instruction ends a super block and one of the special registers is used in the super block, instructions to restore these special registers are inserted. If the current instruction ends a super block and the super block contains instructions that may cause a trap, a trap barrier is inserted. If the current instruction is a branch, these instruction are always inserted before the branch. Otherwise, these instructions are inserted after the current instruction, along with a return instruction that transfers control to the main loop in the thread execution routine. If the last instruction is a branch, this return instruction is already part of the instruction sequence that replaces the branch during modification of instructions.

In the second step, all instructions are modified based on the information in the configuration files. To this end, all super blocks in the procedure are processed sequentially. Furthermore, all basic blocks that belong to the current super block and all instructions that belong to the current basic block are processed sequentially as well.

Modifying Instructions. Each instruction is modified in the following way: First of all, either the original or modified templates from the corresponding entry in the configuration file are chosen in order to modify the current instruction. The modified templates are only used for branch instructions at super block boundaries as well as special instructions defined by the assembler converter.

After the appropriate set of templates has been chosen, the individual instruction templates are transformed into instructions by substituting all keywords with the actual values from the current instruction. The individual keywords have already been described in Section 3.5.1. Afterwards, the current instruction is replaced by the list of transformed instructions.

In the third step, the individual instructions are printed to the output file. All instructions in the procedure are processed in the original sequence in order to maintain program semantics without a re-layout of the basic blocks. The instructions are transformed into text strings via the styles defined in the platform-specific configuration file.

3.5.9 Statistics

The code conversion process associated with emulated multithreading modifies all instructions that belong to internal procedures. In order to evaluate the impact of these modifications, the assembler converter maintains two different sets of detailed statistics about the conversion process: The first set

covers the original instruction sequence, i.e. before the code conversion process. The second set covers the modified instruction sequence, i.e. after the code conversion process. The statistics for the original instructions are gathered after the final shape of basic and super blocks has been determined, i.e. after the processing of external calls. The statistics for the modified instructions are gathered during the code conversion process.

Depending on the selected level of statistics, information about both statistics is inserted into the modified assembler code after each basic block, super block, procedure, module, or program. Note that the gathered statistics themselves do not depend on the selected level, only the amount of statistics messages is affected.

The assembler converter uses counters to maintain both kinds of statistics on the five different levels mentioned above. Each level provides a separate set of counters to record the individual events on this level. If the current basic block, super block, procedure, or module is finished, the contents of the individual counters are added to the corresponding counters on the next level and cleared afterwards. For example, if the current basic block is finished, the statistics for this block are added to the statistics for the current super block and are cleared afterwards in order to process the next basic block.

The assembler converter provides a rich set of events that can be easily extended by adding counters and inserting corresponding calls to update these counters into the assembler converter. The following events are supported:

- The number of read accesses for each integer and floating-point register.
- The number of write accesses for each integer and floating-point register.
- The number of instructions in the current basic block, super block, procedure, module, or program.
- The number of instructions of a given type in the current basic block, super block, procedure, module, or program.
- The number of basic blocks in the current super block, procedure, module, or program.
- The number of super blocks in the current procedure, module, or program.
- The number of procedures in the current module, or program.
- The number of modules in the current program.

The assembler converter partitions the individual instructions to one of several groups. The assignment is based on keywords in the type field of the corresponding entries in the platform-specific configuration file. The INT_MEM keyword designates the instruction as a memory integer instruction, while the INT_CTRL keyword designates the instruction as an integer control instruction. The INT_ARITH and INT_LOGIC keywords are used for integer arithmetic and logical instructions, respectively. Byte and word instructions are designated by the INT_BYTE keyword, while multimedia instructions are designated by the INT_MEDIA keyword. The FP_MEM, FP_CTRL, and FP_ARITH keywords are used to designate floating-point

memory, control and arithmetic instructions, respectively. Last but not least, the INT_MISC keyword is used for miscellaneous instructions.

3.6 Register Partitioning

Partitioning the register set and confining the individual threads to their corresponding partitions allows emulated multithreading to omit all save and restore operations to and from the thread descriptor upon a context switch. Partitioning impacts the performance of emulated multithreading in two ways: On one hand, the omitted save and restore operations will reduce the context switch overhead. On the other hand, each thread has only a limited number of registers available, probably increasing the number of register spills. In addition, the small number of threads that can be used with register partitioning may impact the ability to hide the latency of remote memory accesses in massively parallel processors.

Support for partitioning requires changes to the high-level language converter, the assembler converter, as well as the emulation library. This section covers only the modifications that have to be made to the assembler converter, since the modifications for the high-level language converter and the emulation library have already been described in Sections 3.3 and 3.4, respectively.

In the case of the assembler converter, only a few modifications are necessary: First of all, the individual registers in the register set have to be assigned to different partitions. Note that each partition requires its own version of the emulation registers and the global pointer, i.e. four registers in total. The emuvar optimization is useful in conjunction with register partitioning as it reduces the number of emulation registers to one, i.e. the FramePtr register that contains the address of the current thread descriptor.

The assembler converter maintains an internal database that contains all registers, hence it is sufficient to update this database with the corresponding partition assignments. Access to the register database is provided by a set of routines that take the partition number into account for every access to the database.

The parser inserts a partition-specific prolog instruction in front of each procedure and therefore has to determine the partition for the current procedure. This is accomplished by examining the name of the procedure, i.e. the global label, as the high-level language converter adds a general prefix as well as a partition-specific postfix string to the name of all duplicated procedures.

The assembler converter determines the partition for the current procedure in the same way before the code conversion process is started. The number of the partition is stored in the procedure data structure, hence all routines in the assembler converter have access to the partition number.

Enabling partition support actually simplifies the conversion process, as the individual live ranges must no longer be splitted across super block bound-

aries. Hence the call to the split_graph() routine can be omitted if partitioning is enabled. Recall that the save and restore instructions before and after a context switch are no longer required, hence the live ranges can be extended across super block boundaries.

Calls to external procedures present a problem in combination with partitioning: The call prologue and epilogue have to use the registers as specified by the calling convention and can therefore not be confined to a partition. If one of the threads calls an external procedure, the call prologue, epilogue and the callee itself will destroy the contents of registers in other partitions. The partition of the thread that calls the external procedure will not be affected, as this situation is already handled by the processing of external calls as described in Section 3.5.5.

In order to protect the other partitions, all registers have to be saved, e.g. these partitions have to be saved directly before the call prologue and restored right after the end of the call epilogue. Note that the registers can not be saved to the individual thread descriptors as the position of the other threads at the point of the external call cannot be determined at compile-time. Hence it is unknown which registers are actually used and how the registers were allocated. For these reasons, storage for the whole register set has to be provided, e.g. by using a global thread descriptor. Note that only one external call is executed at any time, hence one copy of the register set is sufficient. The save and restore to this storage area is accomplished by inserting corresponding save and restore instructions before and after the call.

A drawback of this approach is the impact on performance due to the large number of save and restore operations before and after the call. Calls to external procedures should therefore be avoided if register partitioning is used.

3.7 Platform

The implementation of emulated multithreading, i.e. the architecture of the high-level language converter, the assembler converter, and the emulation library is described in the previous sections. So far, this description did not cover platform-specific issues. While most parts of the implementation applies to all platforms, there are some platform-specific issues. The following paragraphs describe these issues for the high-level language converter, assembler converter, and emulation library, respectively.

The high-level language converter operates on the high-level language source code and is therefore largely platform independent. However, the project-specific configuration files that are created by the converter are platform-specific: Recall from Section 3.5.1 that each entry in the configuration file specifies the required, optional, arguments, and return registers for a specific internal or external procedure. These registers are gathered by

translating the declaration of the corresponding procedure according to the calling conventions. As these calling conventions depend on the processor architecture and the operating system, i.e. are platform-specific, the translation process has to be adapted to the different calling conventions. This is accomplished by isolating the translation routine in a platform-specific source file and defining the names of the individual registers in a platform-specific header file.

There are several platform-specific issues in the assembler converter: The lexer and parser used to process the assembler source are platform-specific as the syntax of the assembler source depends on the processor architecture and the operating system, i.e. the assembler. The lexer must be able to recognize the individual tokens, especially identifiers, instruction, directives, and registers. As the namespace of these tokens overlaps partially, the lexer has to maintain a list of all instructions, directives, and register names in order to distinguish identifiers from the other tokens. The lexer definition and a corresponding header file are therefore platform-specific.

Recall that the assembler converter maintains an internal register database that contains the names and characteristics of all registers. As the information in this database depends on the calling convention, i.e. the processor architecture and the operating system, this database is platform-specific as well. The use of this database allows the lexer and parsers for the platform- and project-specific configuration files to be general, as all register names are translated via the register database.

Recall that the assembler converter uses two different passes over the assembler source. As the structure of the assembler source depends on the processor architecture and the operating system, the individual lexer and parser definitions are platform-specific. Apart from these platform-specific issues, the second pass is identical on all platforms, i.e. performs the same set of actions. However, the first pass performs platform-specific actions: The assembler converter for the Alpha processor architecture under the Tru64 operating system uses the first pass to detect internal procedures, i.e. the corresponding global labels. If such a global label is encountered, the first local label after the end of the procedure prolog is stored in the list of internal procedure names. This information is used to handle an optimization of the compiler used in the Tru64 operating system: If a procedure does not use the stack, i.e. a procedure prolog is not necessary, the compiler uses the first local label after the end of the procedure prolog instead of the procedure name in all calls to this procedure. Note that the procedure prolog is created in all cases. As the local label is usually designated by a four-digit string, the assembler converter has to associate these local labels with the corresponding global label, i.e. the name of the procedure, in order to determine the proper entry point to the procedure. Note that the code conversion process must always begin at the procedure entry point, otherwise the save and restore

instructions may be placed incorrectly, especially if super block optimization is used.

Apart from detecting internal procedures, all procedure calls are detected and the called label is marked accordingly. After parsing the whole source file, all pairs of local and global labels are instructed and those labels that were called at least once are used during the second pass to detect internal procedures. Note that either one of the two labels in a pair will be used, but not both.

Another platform-specific issue is the use of relocation operands: The compiler used in the Tru64 operating system specifies the relocation type explicitly, these relocations are translated into the corresponding offsets by the assembler. For example, the following code sequence is used to initialize the global pointer, i.e. the pointer to the global segment that contains all global variables.

```
ldah    $gp, ($27)!gpdisp!1
lda     $gp, ($gp)!gpdisp!1
```

Note that the type of the relocation, i.e. global displacement (gpdisp), and a unique number is appended to the instruction, separated by exclamation marks. The unique number is used to link those instructions that belong to the same relocation operation, and is ordered by the increasing line numbers of the assembler source. The relocation assumes that the base register contains the address of the instruction itself, otherwise the offset will be miscalculated. Similar instruction sequences are used to load global variables and constants.

There are several issues with relocation operands: First of all, the instruction sequences that replace branch instructions at the end of super blocks use relocation sequences to load the address of the target instruction into a register. The insertion of additional relocation sequences destroys the ordering of the original relocation sequences, i.e. the corresponding relocation numbers must be updated. This is accomplished by updating the relocation numbers during instruction update: Recall that instructions are updated in the same order as they appear in the original source file, i.e. it is sufficient to maintain the number of additional relocation sequences inserted so far and update all other relocation sequences accordingly. All instructions that belong to the same relocation sequence are linked and can therefore be updated at the same time.

There is an additional restriction regarding relocation sequences: The corresponding offsets calculated by the assembler are only correct if the base register used in the relocation sequence is identical to the destination register of the indirect branch that produces the return address. Otherwise the calculated offset is off by 4 bytes times the number of instructions between the indirect branch and the start of the relocation sequence. Unfortunately, the conversion process reallocates all registers and introduces new instructions, e.g. save and restore instructions, hence it is likely that the assembler

will not be able to calculate the offsets correctly. Therefore the offsets used in the relocation sequences are updated by counting the number of instructions between the indirect branch and the start of the relocation sequence and multiplying the result by four. This process is performed after all instruction sequences have been updated, i.e. after all save and restore instructions have been inserted into the assembler source.

Another issue with relocation sequences is the use of these sequences to restore the global pointer after every procedure call, as the global pointer resides in a register that is not callee-save. Normally the reload operation is performed by a single lda/ldah instruction pair right after the call instruction. However, instruction scheduling may rearrange the instructions such that there are one or more instructions between the call itself and the start of the relocation sequence as well as between the two instructions in the sequence. Note that these instructions can even be moved across basic block boundaries. As all threads share the global pointer, all updates to the corresponding register must be atomic, i.e. there must not be a super block boundary between the two instructions of the relocation sequence. This problem is handled by moving the second instruction right after the first instruction in these cases.

Another problem is the base register of the first instruction in the relocation sequence: As mentioned above, this register must be identical with the destination register of the call instruction, otherwise the corresponding offsets will be miscalculated by the assembler. This problem is handled by using a call epilogue that extends from the actual call instruction to the first instruction of the relocation sequence and adding the corresponding register to the call mask. Recall that all registers in the callmask are allocated to themself inside the call prologue and epilogue. For similar reasons, the register that contains the address of the callee is added to the call mask as well, as the calling convention of the Tru64 operating system requires that register r25 is used for this purpose.

The emulation library is completely platform-specific, otherwise most of the routines perform similar actions: The thread attribute routines are identical across all platforms, while the thread argument and thread creation routines depend on the calling conventions of the corresponding operating system. The thread execution routine is written in assembler, hence depends on the syntax of the corresponding assembler, otherwise the thread execution routine is identical across all platforms. The same applies to the EMUthread_cswap(), EMUthread_self(), and EMUthread_barrier() routines, although the barrier routine for the Cray T3E calls the system-wide barrier once all threads on the corresponding processor have entered the barrier.

The E-register routines are implemented in very different ways: On the Cray T3E, these routines access the E-register hardware directly in order to implement the split transaction communication routines. The individual routines are covered in large detail in Appendix B. On the Compaq XP1000, these routines are replaced by macros that load and store values into a static

array that represents the E-registers. For example, the EMUereg_int_get() instruction loads the content of the specified address and stores it in the entry of the array, while the routine returns the entry of the array. Note that this approach only works on single-processor systems or multi-processors systems using symmetric multiprocessing, i.e. shared memory.

3.8 Compiler Integration

The high-level language and assembler converters described above use various techniques that are similar to the ones used in compilers, e.g. control and data-flow analysis, register allocation. The integration of these converters into a compiler is therefore plausible. Integrating the converters into a compiler would have several benefits:

The compiler provides access to information that has to be recovered from the high-level language or assembler source at present. Most tasks of the high-level language converter are already performed by the compiler frontend, like constructing the control-flow graph and identifying the type and location of procedure calls. The high-level language converter is therefore reduced to a single pass over the call graph that creates the module-specific configuration file. The assembler converter can use additional information, e.g. profiling, for new optimizations, e.g. during creation of super blocks. In addition, the limitations of the assembler converter with respect to variable argument procedures and procedure values could be resolved.

The code conversion process could be improved, as the interaction between different compiler phases usually has a positive impact on the quality of the generated code, e.g. for register allocation and instruction scheduling [BEH91]. In a similar way, the other phases of the compiler could be steered to produce code that is beneficial to emulated multithreading. In addition, new or improved optimizations are possible as well, e.g. optimizations that require a re-layout of the code.

Integration also provides access to the existing compiler infrastructure, especially optimization phases. Some of these optimizations could be repeated on the converted code, e.g. instruction scheduling and code layout. In the current stand-alone implementations, these tasks are left to the assembler. The existing infrastructure would also ease the implementation of the converter, since common components have no longer to be built from scratch.

The converters can be more easily ported to new languages and platforms: Once the converters are integrated into the compiler, all high-level languages already supported by the compiler are supported by emulated multithreading as well. In order to support a new high-level language, adding a new compiler frontend, which is independent of the converters, is sufficient. Similar, emulated multithreading should be ported easily to all platforms that the compiler supports, since the converters operate on internal representations instead of platform-specific assembler code.

Based on the benefits described above, compiler integration would be an important step in the evolution of emulated multithreading. To make such an integration feasible, a compiler system must be found that is modular by design and available in source code. The SUIF2 compiler system [Lam99] satisfies these conditions and is therefore a good candidate for integration with emulated multithreading. The following paragraphs describe the SUIF2 compiler system in detail.

The SUIF2 system is a compiler infrastructure that was developed to support research and development of compilation techniques as part of the National Compiler Infrastructure (NCI) project. The SUIF acronym stands for Stanford University Intermediate Format, the name of the internal representation that is used by the compiler infrastructure. The current implementation of the SUIF system is called SUIF2 [Lam99], a complete redesign from the earlier SUIF1 [WFW+94] implementation.

The SUIF2 compiler architecture contains the following components: intermediate format, kernel, modules, and the compiler driver. The intermediate format is an extensible program representation that is used throughout the compiler to transfer information between different phases of the compiler. The kernel defines and implements the compiler environment, i.e. the program representation, and all modules.

Modules constitute the core of the SUIF system. A module can either be a set of nodes in the intermediate representation or a program analysis pass that operates on the intermediate representation of a program. The compiler is steered by the compiler driver that creates the compiler environment, imports all required modules, loads the internal representation of a program, applies a series of transformations, and creates a new representation. The initial representation is created by one of the compiler frontends, the last representation is used by one of the backends to create actual code. The SUIF2 compiler system currently supports frontends for the C and Fortran languages as well as backends for the Alpha and IA32 architectures. The backends are provided by the machineSUIF project described in the next paragraph.

Similar to SUIF, machineSUIF was developed to support research and development of compiler backends. The machineSUIF system is based on SUIF and contains the following components: virtual machine, programming interface, as well as several libraries. The SUIFvm virtual machine is the internal machine-independent representation used by the backends to transfer information between phases. In addition, each backend defines a machine-dependent representation. The optimization programming interface is a machine-independent interface to certain optimization passes that operate either on the SUIFvm representation or on a machine-dependent representation. Machine-independent libraries support manipulation of control-flow graphs, control-flow and data-flow analysis, while machine-dependent

libraries add support for specific targets and implement the optimization programming interface.

The typical backend flow consists of several steps: First of all, the intermediate representation in the SUIF format is lowered to the SUIFvm representation by generating code for this virtual machine. Several transformations and optimizations are performed on the intermediate representation at this level. The representation is subsequently realized, i.e. transformed into code for the actual target platform. Again, several optimization phases are performed on this level. Note that the two different intermediate representations are only dialects of the same intermediate representation, i.e. the same optimization passes can be used in both cases. After all optimizations have been performed at this level, the code is generated and either printed as an assembler source file for use by an external assembler or directly assembled into an object file.

The high-level language and assembler converters can be integrated into the SUIF system by reimplementing the high-level language converter as a SUIF optimization pass and the assembler converter as a machineSUIF optimization pass. Although integration into the SUIF compiler structure seems feasible, it was not an option for the current implementation of the converters: The SUIF system was not mature enough at the time the fundamental design of the converters was chosen. For example, the optimization programming interface was totally undocumented, hence integration into the SUIF system seemed too great a risk. However, the next implementation of emulated multithreading should be integrated into the SUIF system in order to realize the benefits mentioned at the beginning of this section.

4. Benchmarks

In order to investigate the characteristics of emulated multithreading, benchmarks are used in the evaluation of emulated multithreading . To make this evaluation as useful as possible, these benchmarks should satisfy the following conditions:

- The benchmarks should cover a wide range of computational problems and program characteristics in order to provide a solid base for the evaluation.
- The benchmarks should be widely used and should have well-known characteristics. This facilitates comparisons with other approaches and identifying the impact of emulated multithreading on performance.
- Since emulated multithreading was designed to hide latency in massively parallel processors, the benchmarks should be parallel.
- The source code of the benchmarks should be available and be easily portable to all platforms used in the evaluation.
- The benchmarks should have sufficiently small time and space requirements in order to make a thorough evaluation feasible, since the computing time is provided by grants and is therefore limited.

The following sections describe several parallel benchmark suites and determine whether these benchmark suites are suitable with respect to the evaluation of emulated multithreading according to the conditions outlined above. After choosing one of the benchmark suites, the individual benchmarks used during the evaluation of emulated multithreading are described in detail.

4.1 Benchmark Suites

This section describes several popular benchmark suites: The LINPACK benchmark is described in Section 4.1.1, while Section 4.1.2 covers the LFK benchmark suite. Sections 4.1.3 and 4.1.4 cover the ParkBench and NPB benchmark suites, respectively. Section 4.1.5 covers the Perfect Club benchmark suite, while Section 4.1.6 covers the SPLASH2 benchmark suite. Apart from a description of the individual benchmark suites, the suitability of each benchmark suite with respect to the evaluation of emulated multithreading is determined.

4.1.1 LINPACK

The LINPACK [Don90] package is used for solving dense systems of linear equations and consists of a collection of subroutines written in Fortran. The equation systems are solved by decomposition of the corresponding matrix into a product of well-structured matrices. These well-structured matrices are easily solved and the results are combined to solve the original equation system. A popular example of such a decomposition is the LU decomposition, which decomposes the original matrix A into a unit lower-triangular matrix L as well as an upper-triangular matrix U. The equation systems represented by the L and U matrices are easily solved using forward and backward substitution, respectively.

The LINPACK benchmark uses the BLAS (Basic Linear Algebra Subroutines) package. This package contains three different sets of routines named BLAS levels one to three: BLAS level one routines support simple vector-vector operations, while BLAS level two routines support matrix-vector operations and BLAS level three routines support matrix-matrix operations.

The LINPACK benchmark suite consists of three different problems:

- The first problem is to solve a dense system of linear equations represented by a 100×100 matrix. The algorithm uses LU decomposition as well as forward and backward substitution and is based on level one subroutines from the BLAS package.
- The second problem is to solve a dense system of linear equations represented by a 300×300 matrix. Again, LU decomposition and forward and backward substitution is used, but the algorithm is based on level two subroutines from the BLAS package.
- The third problem is to solve a dense system of linear equations represented by a 1000×1000 matrix. The algorithm is based on the level three subroutines from the BLAS package, i.e. the specific algorithm used to solve the equation system depends entirely on the BLAS implementation.

The LINPACK benchmark is quite popular, hence benchmark results for almost every computer system are available. Therefore the LINPACK benchmark is often used to track the evolution of computer performance. However, the LINPACK benchmark is restricted to solving dense systems of linear equations, which makes predictions about performance in other areas difficult at best.

4.1.2 LFK

The Livermore Fortran Kernels (LFK) [McM88] consist of 24 different computation kernels that were taken from scientific applications. Kernels are small but important pieces of code. Note that the kernels are sometimes called the Lawrence Livermore Loops (LLL). All kernels are written in Fortran [MR96] and distributed in the form of a single source file that contains the 24 kernels

as well as benchmark execution and timing support. The individual kernels range from a few lines of code to a few dozen lines of code in size and perform the following operations:

Loop 1 is a fragment from a hydrodynamic code.
Loop 2 is a fragment from a Cholesky-Conjugate gradient code.
Loop 3 computes the inner product of two vectors.
Loop 4 computes banded linear equations.
Loop 5 is a fragment from a tridiagonal elimination routine.
Loop 6 is a general linear recurrence equation.
Loop 7 computes a state equation.
Loop 8 is a fragment from an implicit integration code.
Loop 9 is a fragment from an integrate predictor code.
Loop 10 is a fragment from a difference predictor code.
Loop 11 computes the first sum of a vector.
Loop 12 computes the first difference of a vector.
Loop 13 is a fragment from a two-dimensional particle code.
Loop 14 is a fragment from a one-dimensional particle code.
Loop 15 performs various matrix computations.
Loop 16 is a fragment from a Monte Carlo code.
Loop 17 performs various vector computations.
Loop 18 is a fragment from a two-dimensional explicit hydrodynamic code.
Loop 19 computes a general linear recurrence equation.
Loop 20 is a fragment from a discrete ordinates transport program.
Loop 21 computes a complex vector equation.
Loop 22 is a fragment from a Planckian distribution code.
Loop 23 is a fragment from a two-dimensional implicit hydrodynamics code.
Loop 24 computes the minimum element in a vector.

The parallel complexity of the individual kernels was analyzed by Feo [Feo88]. The Livermore Fortran Kernels themselves do not use explicit parallelization directives, i.e. parallelization and vectorization is the responsibility of the compiler. Due to the small data sizes, the individual kernels do not scale well for larger numbers of processors, i.e. when executed on a massively parallel processor. Another problem with the Livermore Fortran Kernels is the regular structure, e.g. loops of the kernels, which favors techniques like prefetching.

4.1.3 ParkBench

The ParkBench [HB94][DH95] (Parallel Kernels and Benchmarks) benchmark suite was developed for distributed memory machines using message-passing. The benchmark suite consists of several different groups of benchmarks: low-level benchmarks, kernel benchmarks, compact applications, and compiler benchmarks. Since there are no benchmarks in the compact applications

group and the compiler benchmarks are not targeted towards performance evaluation, the following paragraphs describe only the benchmarks in the first two groups. All benchmarks are written in the Fortran 77 language and use the Parallel Virtual Machine (PVM) library [GBD+94] for communication between processors. Note that PVM has been replaced by the Message Passing Interface (MPI) [SOHL+98][GHLL+98], at least for commercial applications.

Low-Level Benchmarks. The group of low-level benchmarks contains single- and multi-processor benchmarks. The set of single-processor benchmarks contains the following benchmarks:

TICK1 measures the resolution of the clock that is used for timing measurements.

TICK2 determines whether the clock that is used for timing measurements measures wall-clock time.

RINF1 measures the execution time of several loops across different vector lengths and computes the corresponding performance in MFLOPS for the largest vector length, as well as the vector length required to achieve 50% of the maximum performance.

POLY1 measures the memory bottleneck between the processor registers and the cache by repeating 1 000 polynomial evaluations across various vector lengths.

POLY2 measures the memory bottleneck between the processor registers and main memory by performing one polynomial evaluation across several vector lengths. As the caches are flushed before each evaluation, the benchmark will operate out of main memory.

The set of multi-processor benchmarks covers communication performance and contains the following benchmarks:

COMMS1 measures the time to send a message of size n between processors.

COMMS2 is similar to the COMMS1 benchmark, but this time both processors send ping-pong messages to each other simultaneously.

COMMS3 is a generalized version of the COMMS2 benchmark, where all available processor participate in sending and receiving messages: Each processor sends messages to all other processors and subsequently receives messages from all other processors.

POLY3 is similar to the POLY1 benchmark, but stores the data that is used during the polynomial evaluations on another processor. The POLY3 benchmark therefore measures the memory bottleneck between the processor registers and remote memory.

SYNCH1 consists of a single barrier synchronization across all available processors and measures the efficiency of these barriers.

The TICK1, TICK2, RINF1, COMMS1, COMMS2, SYNCH1 benchmarks were taken from the Genesis benchmarks suite [Hey91], while the

POLY1 and POLY2 benchmarks were taken from the EuroBench [vdSdR93] and Hockney [Hoc93] benchmarks suites, respectively. The COMMS3 and POLY3 benchmarks were written for the ParkBench benchmark suite.

Kernel Benchmarks. The group of kernel benchmarks consists of four different sets of benchmarks: matrix kernels, Fourier transformations, partial differential equation solvers and miscellaneous.

The set of matrix kernels consists of the following benchmarks:

MM multiplies dense block-partitioned matrices.
MT transposes a dense block-partitioned matrix.
LU decomposes a dense matrix into a unit lower-triangular as well as an upper-triangular matrix using LU decomposition.
QR decomposes a matrix into unitary and upper-triangular matrices.
BT computes a tridiagonalization of a block-partitioned matrix.

The set of Fourier kernels contains the following benchmarks:

FFT1D performs a one-dimensional Fast Fourier Transformation (FFT) by computing the forward FFT on two vectors, multiplying the result and computing an inverse FFT on the result.
FFT3D performs a three-dimensional Fast Fourier Transformation by computing the forward FFT of a three-dimensional matrix, multiplying the result several times by exponential factors and computing an inverse FFT on the result.

The set of partial differential equation kernels contains the following benchmarks:

SOR uses successive over-relaxation (SOR) to solve the Poisson equation on a three-dimensional grid by parallel red-black relaxation with Chebyshev acceleration.
MG uses a multigrid algorithm to determine an approximation to the discrete Poisson problem on a three-dimensional grid with boundary conditions.

The set of miscellaneous kernels contains the following benchmarks:

EP generates pseudo-random floating-point values and counts the number of Gaussian deviates inside various squares around the origin.
CG uses the inverse power method to estimate the largest Eigenvalue of a symmetric, positive-definite sparse matrix.
IS sorts a large array of integers. The particular algorithm is not specified, as the use of vendor-supplied sort routines is allowed. However, the initial distribution of the integer array across the processors is specified.
IO measures I/O related parameters like startup time, bandwidth, and latency. The benchmark is provided in a paper-and-pencil fashion, hence the implementation is left to the benchmarker.

The FFT3D, MG, EP, CG, IS benchmarks are taken from the NAS parallel benchmark suite [BBLS91][BHS+95] described in Section 4.1.4, while the SOR benchmark is taken from the Genesis benchmarks suite [Hey91]. The matrix kernels, FFT1D and I/O benchmarks were written for the ParkBench benchmark suite.

The ParkBench benchmark suite covers a wide range of scientific applications. However, the low-level benchmarks are probably too simple to provide useful insights. The kernel benchmarks are better suited in this regard, but most of these benchmarks are taken from the NAS parallel benchmark suite, hence these benchmark suite could be used instead. Another problem with the ParkBench benchmark suite is the use of message-passing primitives for communication as the emulation library currently supports only shared memory primitives. Therefore porting the benchmarks to the Cray T3E will require a significant amount of work.

4.1.4 NPB

The NAS Parallel Benchmark Suite (NPB) consists of five kernels and three compact applications from the field of computational fluid dynamics. The benchmarks were initially released in paper-and-pencil fashion, i.e. by providing a detailed description of the problems that omitted any implementation-specific details [BBLS91]. A later release of the NPB included implementations for five out of the original eight benchmarks [BHS+95]. These benchmarks are written in the Fortran 77 language and use the MPI standard for communications between processors. Therefore the NPB benchmark suite is highly portable. The suite consists of the following benchmarks:

EP generates pseudo-random floating-point values and counts the number of Gaussian deviates inside various squares around the origin.

CG uses the inverse power method to estimate the largest Eigenvalue of a symmetric, positive-definite sparse matrix.

IS sorts a large array of integers. The particular algorithm is not specified, as the use of vendor-supplied sort routines is allowed. However, the initial distribution of the integer array across the processors is specified.

FT computes a three-dimensional Fast Fourier transformation using a distribution of data around the z-dimension. The benchmark performs a forward three-dimensional FFT as multiple one-dimensional FFTs in the x- and y-dimensions, transposes the corresponding matrix and performs multiple one-dimensional FFTs in the z-dimension.

MG uses a multigrid algorithm to determine an approximation to the discrete Poisson problem on a three-dimensional grid with boundary conditions.

LU decomposes a dense matrix into a unit lower-triangular as well as an upper-triangular matrix using partial pivoting.

SP solves three sets of scalar pentadiagonal systems of equations.

BT solves three sets of block tridiagonal systems of equations.

The NAS parallel benchmark suite uses three different (A, B, C) input sizes for each individual benchmark. However, the last two input sizes are too large to make a thorough evaluation feasible with the limited amount of available computing time. Another problem is the use of message-passing primitives for communication between processors as the emulation library currently supports only shared-memory primitives.

4.1.5 Perfect Club

The Perfect Club benchmark suite is a collection of 13 applications written in the Fortran language. The individual applications originate from various areas of scientific applications:

ADM	simulates pollutant concentration and deposition patterns in lake-shore environments by solving systems of hydrodynamic equations.
ARC3D	analyzes three-dimensional fluid flow problems by solving Euler and Navier-Stokes equations.
BDNA	simulates the molecular dynamics of biomolecules in water using the Biomol package.
DYFESM	analyzes symmetric anisotropic structures using a dynamic two-dimensional finite-element method.
FLO52Q	analyzes the transonic inviscid flow past an airfoil by solving the unsteady Euler equations.
MDG	simulates the molecular dynamics of 343 water molecules in the liquid state at room temperature.
MG3D	is used to investigate the geological structure of the earth using a seismic migration code.
OCEAN	simulates large-scale ocean movements based on eddy and boundary currents by solving the dynamical equations of a two-dimensional Boussinesq fluid layer.
QCD	simulates long-range effects in Quantum Chromodynamics theory.
SPEC77	simulates atmospheric flow using a global spectral model based on solving partial differential equations.
SPICE	simulates the behavior of integrated circuits using non-linear direct-current, non-linear transient and linear alternating-current analysis.
TRACK	computes the position, velocity, and acceleration of a set of targets by observing the targets at regular intervals.
TRFD	simulates two-electron integral transformation based on a series of matrix multiplications.

The individual benchmarks in the Perfect Club benchmark suite cover a wide range of scientific applications. However, the size of the applications, i.e. 50 000 lines of code, makes the porting of the applications infeasible.

4.1.6 SPLASH2

The SPLASH (Stanford ParalleL Applications for SHared memory) parallel benchmark suite [SGL92] was designed to provide a suite of benchmarks for cache-coherent shared-memory multiprocessors that increases the consistency and comparability between different studies. Several limitations of this benchmark suite, namely the limited coverage of algorithms, the limited scaling for larger processor counts, as well as the limited support for caches, led to the release of the SPLASH-2 benchmarks [WOT+95]. The second release addresses these problems and has gained wide-spread acceptance in computer architecture research due to the reasonably small problem sizes that facilitate software simulation for new architectural concepts.

The SPLASH-2 benchmark suite was written for cache-coherent shared-memory systems and uses the PARMACS macro package [BBD+87] for parallelization. The suite consists of four kernels and eight complete applications written in the C [DM91] language:

radix	sorts a large set of integers using an iterative algorithm.
fft	performs a Fast Fourier transformation using a six-step algorithm.
lu	performs a LU decomposition on a dense matrix.
cholesky	performs a Cholesky factorization of a sparse matrix.
barnes	simulates a n-particle system using the Barnes-Hut method.
fmm	simulates a n-particle system using the fast-multipole method.
ocean	simulates large-scale ocean movements based on eddy and boundary currents.
radiosity	calculates illumination in a three-dimensional scene using an iterative hierarchical algorithm.
volrend	renders the volumes of a three-dimensional scene.
water1	calculates molecule dynamics within a system of water molecules using an $O(n^2)$ algorithm.
water2	calculates molecule dynamics within a system of water molecules using an $O(n)$ algorithm.

The algorithms used in the radix, fft, and lu benchmarks are described in [BLM+91], [Bai90], and [WSH94], respectively. The algorithms used in the cholesky, barnes, and fmm benchmarks are described in [RG94], [SHT+95], and [SHHG93], respectively. The ocean, radiosity, and volrend benchmarks are described in [SH92], [SHT+95], and [NL92], respectively. Finally, the algorithms used in the water benchmarks are described in [WOT+95]

4.1.7 Summary

Recall that the benchmark suite used during the evaluation of emulated multithreading should satisfy the five conditions mentioned above. The first condition, i.e. wide coverage of application areas, is satisfied by all benchmark suites except LINPACK and LFK. The LINPACK benchmark covers only

one application area, i.e. solving dense systems of linear equations. The LFK benchmark suite consists of small kernels from many areas of scientific computing, but is limited to regular structures, i.e. loop-level parallelism. Note that all benchmark suites are restricted to scientific applications.

The second condition, i.e. widespread use, is more or less satisfied by all benchmark suites: The LINPACK benchmark is widely used and benchmark results for almost every machine are available. The NPB benchmarks are quite popular for the evaluation of large-scale massively parallel processors. The SPLASH2 benchmarks are widely used in computer architecture research. The third condition, i.e. efficient parallelization, is satisfied by all benchmark suites except LFK and Perfect Club, which are written in Fortran and contain no parallelization directives at all.

The fourth condition, i.e. availability and ease of portability, is only satisfied by the LINPACK, LFK and SPLASH2 benchmark suites. The ParkBench and NPB suites use message-passing communication libraries, e.g. MPI, PVM. This complicates the porting of theses benchmarks, as the emulation library currently supports only shared memory primitives. The Perfect Club suite is too large, i.e. 50 000 lines of code, to make porting of the applications feasible. The fifth condition, i.e. time and space requirements, is satisfied by all benchmark suites except the NPB and Perfect Club suites. These suites define input sizes that are too large to make a thorough evaluation with the limited amount of computing time feasible.

Taken together, the only benchmark suite that more or less satisfies all of these conditions is the SPLASH2 suite. This suite contains kernels and applications from a wide range of scientific applications. Although the benchmarks use the obsolete PARMACS macros for parallelization, the individual benchmarks employ a shared memory model, which makes porting to the shmem and emulation libraries feasible. In addition, most of the benchmarks contain hints for useful distribution of data on distributed memory machines like the Cray T3E. The wide-spread use of these benchmarks in the literature makes comparisons with other approaches feasible. For these reasons, a subset of the SPLASH2 benchmarks is used in the evaluation of emulated multithreading: three kernels and three applications have been ported to the Cray T3E.

4.2 SPLASH2 Benchmark Suite

The evaluation of emulated multithreading is based on three kernels as well as three compact applications taken from the SPLASH2 benchmark suite: The fft, lu, and radix kernels described in Sections 4.2.1, 4.2.2, and 4.2.3, respectively, were selected due to their small code size, ease of portability and the existence of data distribution hints. The ocean application described in Section 4.2.4 was selected due to the regular structure, ease of portability and the existence of data distribution hints. The barnes and fmm application described in Sections 4.2.5 and 4.2.6, respectively, were selected as examples

for irregular applications, although considerable effort is required to port these two applications.

All kernels and applications use the PARMACS macros [BBD+87] for parallelization. Unfortunately, this macro package is obsolete and parallel implementations are not available on the Cray T3E. A sequential implementation of the PARMACS macros is available that allows the execution of such programs on single-processor systems or a single node of a multiprocessor system. However, the benchmarks had to be rewritten in order to replace the PARMACS macros by corresponding routines from the shmem and emulation libraries. In addition, all implicit accesses to remote memory had to be identified and replaced by explicit accesses using the communication routines from the shmem and emulation libraries.

Another difference between the PARMACS macros and the shmem and emulation libraries is the programming model: Using the PARMACS macros, only one processor is running at startup time, the other processors have to be activated explicitly. Using the shmem or emulation libraries, all processors are running at startup time, hence some program sections, e.g. I/O, have to be protected to ensure that only one of the processors executes these sections.

The PARMACS macros used for timing measurements were replaced with calls to the Performance Counter Library (PCL) [BM98] that provides access to the hardware performance counters. These calls use the PCL_CYCLES event, i.e. the cycle counter, to measure wall-clock time. Note that this event is not available on the Cray T3E, hence a per-process cycle counter is used. This represents no problem, as all benchmark experiments were executed in batch mode, i.e. had exclusive access to the processors they were running on.

In order to use emulated multithreading, some modifications have to be made to the original sources. These modifications concern initialization and inter-thread data distribution and are protected by conditional compilation statements, i.e. benchmarks using the shmem and emulation libraries are derived from the same source code.

4.2.1 The FFT Kernel

The FFT (Fast Fourier Transform) is an efficient algorithm for computing the DFT (Discrete Fourier Transform) that is widely used in numerous applications. The next section describes discrete Fourier transformations and largely follows the presentation by Rockmore [Roc00]. The following sections describe the six-step algorithm used by the fft kernel, the specific implementation, as well as the porting details, respectively.

Discrete Fourier Transform. Given an n-element vector X of complex numbers $X = (x_0, \ldots, x_{n-1})$, the discrete Fourier transform computes another n-element vector $Y = (y_0, \ldots, y_{n-1})$, where

$$y_k = \sum_{j=0}^{n-1} x_j w_n^{jk}$$

The $w_n = e^{2\pi i/n}$ are called *roots of unity*, since $e^{2\pi i} = 1$.

The basic idea behind FFT is a transformation of the one-dimensional problem above into a two-dimensional problem. This is accomplished by viewing the n-element vector X as a $n_1 \times n_2$ matrix, where $n = n_1 n_2$. Substituting the indices $j = an_1 + b, k = cn_2 + d$, where $0 \leq a, d < n_2$ and $0 \leq b, c < n_1$, in the equation above yields

$$Y(c,d) = \sum_{b=0}^{n_1-1} w_n^{b(cn_2+d)} \sum_{a=0}^{n_2-1} X(a,b) w_{n_2}^{ad}$$

Now the FFT can be computed using at most $(n_1 n_2)(n_1 + n_2)$ operations by calculating the second term for all values of b, d in at most $n_1 n_2^2$ operations, the remaining transformations in another $n_2 n_1^2$ operations. This approach can be extended to yield the desired $O(n \log n)$ result.

Algorithm. The fft kernel organizes the two n-element vectors mentioned above as two $\sqrt{n} \times \sqrt{n}$ matrices, such that every processor is allocated a contiguous set of rows. The kernel uses the following six-step algorithm as described in [Bai90]:

1. Transpose the data matrix.
2. Perform $\sqrt{n}$ one-dimensional FFTs on the rows of the matrix.
3. Apply the roots of unity to the data matrix.
4. Transpose the data matrix.
5. Perform $\sqrt{n}$ one-dimensional FFTs on the rows of the matrix.
6. Transpose the data matrix.

Communication occurs only during matrix transposition in steps 1, 4 and 6. During matrix transposition, the matrix is divided in patches of size $\sqrt{n}/p \times \sqrt{n}/p$ such that every processor transposes a local patch as well as one patch from every other processor. The matrix transposition steps therefore require all-to-all communication. A barrier synchronization is performed after steps 3, 5, and 6.

Implementation. After parsing the command line, the fft kernel sets various global parameters, e.g. the number of processors, the size of the matrix, as well as the number and size of the cache-lines in the first-level cache. Based on these values, the fft kernel calculates the required matrix size. Row padding is used to ensure that the contiguous array of rows that is allocated to one processor starts on page size boundaries and individual rows start on cache-line boundaries. Afterwards four shared arrays are allocated: x, trans, umain, and umain2. The x and trans arrays hold the data matrix and are used as source and target matrices for transpose operations. The umain2 array holds the roots of unity, while the umain array contains the heavily accessed first row of the umain2 array. After initializing the local and shared arrays, the remaining processors are activated and the Fourier transformation begins. Once the simulation has been completed, these processors are deactivated,

while the first processor gathers timing statistics and checks the validity of the results.

The Fourier Transformation uses the algorithm described above and is implemented in the following way: First of all, the shared umain array is replicated among all processors, a subsequent barrier synchronization is performed to ensure that all processors have finished initialization. Note that each processor is assigned a contiguous set of rows and all processors work only on the assigned rows.

All processor transpose the matrix from the x array to the trans array. Afterwards, each processor performs a one-dimensional FFT and applies the roots of unity to all columns of the trans matrix. A subsequent barrier synchronization ensures that all processors have finished this step before they transpose the matrix from the trans to the x array. Afterwards, each processor performs a one-dimensional FFT on all columns of the trans matrix that belong to it. A subsequent barrier synchronization ensures that all processors have finished this step before they transpose the matrix from the x to the trans array.

The routines that perform the one-dimensional FFT and apply the roots of unity are straight-forward. The transpose routine is more interesting: The source matrix is divided into patches of size $\sqrt{n}/p \times \sqrt{n}/p$. Therefore each processor is assigned p patches, since $\sqrt{n}/p$ rows were allocated to each processor. Every processor transposes one patch locally and gets one patch from every other processor. The transpose of a single patch is performed block-wise to take advantage of spatial locality. The size of the blocks depends on the number of matrix elements per cache-line. Note that the transpose routine is the only part in the fft kernel that requires inter-processor communication.

Porting. Porting the fft kernel described above was straight-forward: First of all, the PARMACS macros were either removed or substituted with calls to the corresponding routines from the shmem, emulation, and performance counter libraries. Some data structures, e.g. barrier structures were removed. All implicit accesses to remote memory were identified and replaced by explicit accesses using communication routines from the shmem and emulation libraries.

The shared arrays were distributed as suggested in the original source, i.e. each processor allocates only those parts of the arrays that belong to it. The umain array was replicated on all processors, hence the replication at the start of the transformation could be removed. The initialization of the distributed arrays was parallelized as well, each processor initializes only those parts of the arrays that belong to it. Apart from the transpose routine, the other functions are basically unchanged. Inside the transpose routine, the transpose of individual matrix elements is now performed with communication routines from the shmem or emulation libraries.

In the case of emulated multithreading, the data distribution remains unchanged, but the work is distributed among the $p \cdot t$ threads on the p pro-

cessors in the same way as if $p \cdot t$ processors were present. Apart from thread initialization, this is the only major change required to support emulated multithreading. Since all processors are active at startup, some changes were made to the initialization routines to ensure that some code sections, e.g. I/O, are only executed by a single processor.

4.2.2 The LU Kernel

The lu kernel decomposes a symmetric, positive-definite matrix A into a unit lower-triangular matrix L and an upper-triangular matrix U such that $A = L \cdot U$ holds. A matrix is symmetric if the matrix is identical to its transposition, i.e. $A = A^{\mathrm{T}}$. A matrix is positive-definite, if $x^{\mathrm{T}} Ax > 0$ holds for all $x \neq 0$. It can be shown that any symmetric positive-definite matrix is invertible.

The decomposition of a matrix A into matrices L, U is useful for solving systems of linear equations. Given a set of linear equations

$$a_{i,0}x_0 + a_{i,1}x_1 \ldots a_{i,n-1}x_{n-1} = b_i \qquad i = 0, \ldots n-1$$

these equation system can be rewritten as $Ax = b$, where $A = (a_{ij})$ and $b = (b_i)$. Given a decomposition of the matrix A into the matrices L, U as described above, this equation can be rewritten as $LUx = b$. Due to the regular structure of the L and U matrices, this equation can be easily solved in two steps: In the first step, the equation $Ly = b$ is solved by forward substitution. In the second step, the equation $Ux = y$ is solved by backward substitution. The following sections describe the decomposition algorithm used by the lu kernel, the implementation of the lu kernel, as well as the porting process, respectively.

Algorithm. The lu kernel uses Gaussian elimination to compute the LU decomposition of a symmetric, positive-definite matrix A. Gaussian elimination is a recursive process that uses up to n steps to compute the decomposition, where n is the size of the matrix. In the ith step, multiples of the ith row are subtracted from all rows below such that the ith variable (x_i) is removed from the corresponding equations. This process is continued until the remaining matrix A has upper-triangular form. The matrix L is created from the individual factors used during the elimination process.

Formally, let $A = (a_{ij})$. Then A can be written as

$$A = \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix}$$

where $A' = (a'_{ij})$ is a $(n-1) \times (n-1)$ matrix, $v = (a_{i1})$, $w = (a_{1i})$ are $(n-1)$-element vectors. The above equation can be rewritten as

$$A = \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix}$$

Note that the term vw^{T}/a_{11} is the outer product of v and w^{T} divided by the scalar a_{11}, i.e. a matrix of size $(n-1) \times (n-1)$. The matrix A' can be recursively decomposed into matrices L', U', which yields the desired decomposition of the matrix A:

$$A = \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix}$$

The lu kernel uses a blocked version of Gaussian elimination that is described below.

Implementation. After parsing the command-line, the lu kernel sets the number of processors p, the size of the matrix n, as well as the block size b. Afterwards the number of rows and columns per processor as well as the number of blocks per row and column is calculated. The individual blocks of the matrix are distributed across the processors in cookie-cutter fashion, i.e. block (i, j) of the matrix belongs to processor

$$j \cdot (\text{number of rows}) + i \bmod (\text{number of columns})$$

Instead of using a two-dimensional array, the matrix A is stored in a four-dimensional array: The first two dimensions specify the blocks of the matrix, while the last two dimensions specify the elements inside the block. The four-dimensional matrix is realized as two levels of two-dimensional arrays, where each element in the first level array contains the address of the corresponding block, i.e. the second-level array of matrix elements.

All blocks that belong to a processor are allocated from a contiguous block of memory and are aligned on page and cache-line boundaries. After initializing the first two-dimensional array with the address of the corresponding blocks, the individual blocks are initialized to form a symmetric, positive-definite matrix. Afterwards the other processors are activated, such that the decomposition algorithm described in the next paragraph is executed by all p processors. Once the decomposition is finished, these processors are deactivated, while the first processor gathers timing statistics and checks the validity of the result.

The decomposition algorithm used by the lu kernel operates on whole subblocks of the matrix instead of single rows. The algorithm consists of a single loop, the number of iterations is equal to the number of blocks per row and column. During each iteration, the following three steps are performed:

- In the first step, the current diagonal block is factorized by the corresponding processor. The diagonal block is factorized column-wise using Gaussian elimination: For each column k of the block, the entries below the diagonal are divided by the diagonal element, the elements of the row k right of the diagonal are subtracted from all rows j below k scaled by the corresponding element of column k:

```
for (k=0; k<n; k++) {
  for (j=k+1; j<n; j++) {
    a[k+j*stride] /= a[k+k*stride];

    daxpy(&a[k+1+j*stride], &a[k+1+k*stride],
          n-k-1, -a[k+j*stride]);
  }
}
```

The resulting factorization is stored in the diagonal block itself. Note that the L, U matrices contain no overlapping entries if the unit diagonal of matrix L is stored implicitly. A subsequent barrier synchronization ensures that none of the other processors proceeds with factorization until the diagonal block has been factorized.

- In the second step, the blocks in the current row and column are updated by the corresponding processors. The blocks in the current column are updated accordingly: For each column k of the block, the elements from row k are subtracted from all rows j below k scaled by the corresponding element of column k:

```
for (k=0; k<dimk; k++) {
  for (j=k+1; j<dimk; j++) {
    daxpy(&a[j*stride_a], &a[k*stride_a],
           dimi, -diag[k+j*stride_diag]);
  }
}
```

The blocks in the current row are factorized column-wise using Gaussian elimination: For each column k, the entries of the column are divided by the corresponding diagonal element of the diagonal block. The elements of the row k right of the diagonal are subtracted from all rows j below k scaled by the corresponding element of the diagonal block:

```
for (k=0; k<dimi; k++) {
  for (j=0; j<dimj; j++) {
    c[k+j*stride_c] /= a[k+k*stride_a];

    daxpy(&c[k+1+j*stride_c], &a[k+1+k*stride_a],
          dimi-k-1, -c[k+j*stride_c]);
  }
}
```

The updated row and column blocks are used during factorization of the interior blocks in the third step. A subsequent barrier synchronization ensures that none of the other processors proceeds with factorization of the interior blocks before the row and column blocks have been factorized.

- In the third step, the remaining interior blocks are updated by the corresponding processors. The interior blocks are updated column-wise according to the factorization of the current diagonal block: For each column k of the interior block, the elements of row k in the corresponding row block are subtracted from all rows in the current block, scaled by the negative of the corresponding entry in row k of the corresponding column block:

```
for (k=0; k<dimk; k++) {
  for (j=0; j<dimj; j++) {
    daxpy(&c[j*stridec], &a[k*stridea],
          dimi, -b[k+j*strideb];);
  }
}
```

Porting. Porting the implementation of the lu kernel described in Section 4.2.2 was straight-forward: All PARMACS macros were either removed or substituted with calls to the corresponding calls from the shmem, emulation, and performance counter libraries. All implicit accesses to remote memory were identified and replaced by explicit accesses using communication routines from the shmem and emulation libraries. Since all processors are active at startup, some changes had to be made to the initialization routines to ensure that some code sections, e.g. I/O are only executed by a single processor.

The matrix A is distributed across the processors such that each processor holds only the blocks of the matrix that belong to him. In this way, the target of a remote memory access is identical to the owner of the corresponding block. In the case of emulated multithreading, the individual blocks of the matrix are distributed among the $p \cdot t$ threads on the p processors in the same way, as if $p \cdot t$ processors were present. Apart from initialization, this is the only major change required to support emulated multithreading.

4.2.3 The Radix Kernel

The radix kernel sorts integer keys using a parallel version of counting-based radix sort. The radix kernel expects three arguments: the number of keys, the size of the radix, and the size of the maximum key. The default values for these arguments are 256 K, 1024, and 512 K, respectively. The following sections describe the algorithm used by the radix kernel, the implementation, as well as the porting process, respectively.

Algorithm. The radix sort algorithm relies on the interpretation of the keys as m-bit integers, where m is the base-two logarithm of the size of the maximum key. Each key is divided into blocks of r bits, where r is the base-two logarithm of the radix size. The algorithm sorts the keys by iterating $\lceil m/r \rceil$ times through a loop, each time sorting the keys according to the current block of r bits. These intermediate sorts must be stable, i.e. the input ordering must be preserved for all keys with equal value in the current block.

Otherwise the intermediate sort destroys the work of previous intermediate sorts. The intermediate sort determines the rank of all keys according to the current block and permutes the keys correspondingly afterwards.

The keys are distributed to the processors such that each processor holds n/p local keys, where n, p is the number of keys and processors, respectively. Each processor maintains a vector index with 2^r elements to store the index and rank of the keys with the corresponding block values. The rank of the individual keys is determined in several steps:

- The 2^r elements of the index vector are cleared. Afterwards the histogram of the local keys is computed and stored in the index vector, i.e. the ith element of the vector contains the number of local keys for which i is the value of the current block.
- For all possible block values k, $0 \leq k \leq 2^r - 1$, the global rank of the first local key with block value k is determined and stored in the k-th element of the index vector. The global rank is the sum of the number of keys on all processors with block values less than k plus the number of keys with block value k on all processors with a smaller processor id. The first summand is determined by maintaining an offset and adding the global sum of the kth elements of the index vector in each iteration. The second summand is determined by a parallel prefix computation on the kth elements of the index vector on all processors.
- After the global rank of the first local key has been determined for all possible block values, the global rank of the remaining keys is determined: The global rank of a local key with block value k is the sum of the global rank of the first key with block value k and the number of local keys with block value k encountered so far. The local keys are processed in sequential order to ensure the stability of the intermediate sorting step.

The radix sort algorithm has several advantages: The algorithm is easy to code and maintain, has a good running time in practice, and performs well on short keys. The main disadvantage of the algorithm is the space requirement, i.e. radix sort does not sort in place. However, radix sort requires less memory than other sorting algorithms, e.g. sample sort [BLM$^+$91].

Implementation. After parsing the command-line, the radix kernel initializes the following parameters: the number of processors p, the number of keys n, the size of the radix r, as well as the size of the maximum key M. Afterwards local and shared memory is allocated and initialized: The key_partition array is a shared array with $p + 1$ elements that is initialized such that the ith element of the array contains the number of the first key allocated to processor i. Note that the pth entry contains the total number of keys. The rank array is a shared array with $p \cdot (r+1)$ elements that contains the rank arrays for all p processors. The first p elements of the rank array are initialized such that the ith entry contains the address of the rank array at processor i . The rank_ff array is a local array with r elements that is used as a local

rank array. The key0, key1 arrays are shared arrays with n elements that contain the actual key values and are used as source and destination during the intermediate sorts, as radix sort does not sort in place.

After the local and shared arrays have been initialized, the remaining $p-1$ processors are activated, such that all p processors execute the parallel sorting algorithm. After all processors have finished sorting the keys, these processors are deactivated again, while the first processor gathers timing statistics and checks the correctness of the result.

Upon startup of the parallel sort, each processor uses a pseudo-random number generator to fill the corresponding part of the key0 array. A barrier synchronization ensures that all processors have finished key generation before the actual sorting begins. As already mentioned above, the parallel sort algorithm consists of a single loop that is traversed $\lceil m/r \rceil$ times, each time sorting the keys according to a block of r bits. During each iteration, the following steps are performed:

- First of all, each processor creates a histogram of all keys that belong to it. This histogram is created in the following way: The part of the global rank array that belongs to the corresponding processor is cleared. Afterwards all local keys are processed by extracting the value of the current block of r bits and incrementing the corresponding entry in the global rank array. In addition, each processor stores the distribution of its keys in the local key_distribution array, i.e. the ith element of the array contains the sum of all elements below i and the ith element of the rank array.
- The local histograms are now merged into a global histogram. This is done in two steps: In the first step, a parallel prefix computation is used to accumulate the local histograms. In the second step, the global histogram is broadcasted to all processors in a reverse parallel prefix operation. These parallel prefix computations use a global array of $2p$ prefix nodes where each node contains two arrays with r elements for distribution and rank, respectively.
  The parallel prefix computation is implemented by building a binary tree on the p processors. Note that the total number of nodes in this tree is given by
  $$\sum_{i=0}^{\log(p)} \frac{p}{2^i} = p \cdot \sum_{i=0}^{\log(p)} (\frac{1}{2})^i \leq 2p$$
  hence the $2p$ elements in the prefix array. Each prefix node contains a lock variable that is used to ensure proper updates of the individual nodes. As only a small subset of processors is used in the upper levels of the tree, computation of the global histogram does not scale well.
- The parallel prefix computation is completed as each processor determines the global rank of the local keys by adding the ranks from all nodes to his left.

- Each processor permutes the local keys from the source to the destination arrays according to the global ranks computed in the previous step. Afterwards the source and target arrays are exchanged unless the current iteration is the last one.

Porting. Porting the implementation of the radix kernel described above was straight-forward: All PARMACS macros were either removed or substituted with calls to the corresponding calls from the shmem, emulation, and performance counter libraries. The shared arrays were distributed as suggested in the original source: The key and rank arrays were distributed such that each processor stores only the local keys. The prefix-tree array used during parallel prefix computation is replicated on all processors and augmented by an additional structure that maps the individual nodes in the prefix tree to processor numbers. Note that the contents of a given prefix node are only valid on the corresponding processor.

All implicit accesses to remote memory were identified and replaced by explicit accesses using the communication routines from shmem or emulation libraries. In the case of emulated multithreading, the individual blocks of the matrix are distributed among the $p \cdot t$ threads on the p processors in the same way, as if $p \cdot t$ processors were present. Apart from initialization, this is the only major change required to support emulated multithreading.

4.2.4 The Ocean Application

The ocean application studies the role of mesoscale eddies and boundary currents in large-scale ocean movements. The ocean is modeled as two different layers, i.e. the lower and the upper layer. The upper layer is driven by wind stress from the overlying atmosphere. In addition, the influences of bottom friction, i.e. between the ocean floor and the bottom layer, as well as lateral friction, i.e. between both layers, are considered. Both layers are modeled as a cuboidal grid. For each grid point, data is recorded at the middle of both layers as well as at the interface between both layers. The distance between grid-points determines the accuracy of the simulation.

Simulation of this wind-driven ocean basin proceeds in several timesteps until a steady state between the eddy currents and the mean ocean flow is reached. In each timestep, the following system of partial differential equations has to be solved [Hol78]:

$$\begin{aligned}
\frac{\delta}{\delta t}\nabla^2\Psi_1 &= J(f + \nabla^2\Psi_1, \Psi_1) - (f_0/H_1)w_2 + F_1 + H_1^{-1}\mathrm{curl}_z\tau \\
\frac{\delta}{\delta t}\nabla^2\Psi_3 &= J(f + \nabla^2\Psi_3, \Psi_3) - (f_0/H_3)w_2 + F_3 \\
\frac{\delta}{\delta t}(\Psi_1 - \Psi_3) &= J(\Psi_1 - \Psi_3, \Psi_2) - (g'/f_0)w_2
\end{aligned}$$

where ∇ and J are the Laplacian and Arakawa Jacobian operators, respectively. H_1, H_3 are the height of the upper and bottom layers, while F_1, F_3 are the lateral friction terms for the upper and bottom layers, respectively. τ is the wind stress influencing the upper layer, $\text{curl}_z\tau$ is the vertical component of the wind-stress curl, and w_2 is the vertical velocity at the interface of the two layers. f and g' represent the Coriolis parameter and the gravity, respectively. The Ψ_1, Ψ_2, Ψ_3 are the stream functions at the middle of the upper layer, the interface between the two layers, and the middle of the bottom layer, respectively.

Algorithm. The algorithm used in the ocean application uses the first difference form of the equations presented above to yield a numerical solution. In each time step, the ocean applications solves the following equation system [Hol78]:

$$\left(\nabla - \frac{Hf_0^2}{H_1H_3g'}\right)\left(\frac{\delta}{\delta t}(\Psi_1 - \Psi_3)\right) = \gamma_a$$

$$\nabla\left(\frac{\delta}{\delta t}\frac{(H_1\Psi_1 + H_3\Psi_3)}{H}\right) = \gamma_b$$

under the condition that

$$\iint \Psi\delta x\delta y = 0 \qquad \text{and} \qquad \frac{\delta}{\delta t}\frac{(H_1\Psi_1 + H_3\Psi_3)}{H} = 0$$

holds. Note that the latter condition applies only to the boundaries.

The right-hand sides of the above equations, i.e. γ_a, γ_b, are given by:

$$\gamma_a = J(f + \nabla(\Psi_1), \Psi_1) - J(f + \nabla(\Psi_3), \Psi_3) - \left(\frac{Hf_0^2}{H_1H_3g'}\right)^2 J(\Psi_1 - \Psi_3, \Psi_2) + H_1^{-1}\text{curl}_z\tau + F_1 - F_3$$

$$\gamma_b = \frac{H_1}{H}J(f + \nabla(\Psi_1), \Psi_1) + \frac{H_3}{H}J(f + \nabla(\Psi_3), \Psi_3) + H_1^{-1}\text{curl}_z\tau + \frac{H_1}{H}F_1 + \frac{H_3}{H}F_3$$

Let Ψ_a, Ψ_b be stream functions that satisfy certain boundary conditions, then

$$\Psi = \Psi_a + \frac{\iint \psi_a\delta x\delta y}{\iint \psi_b\delta x\delta y}\Psi_b$$

is used to aid the solution of the equations above. The algorithm used in the ocean application arranges the above computations such that the computations can be efficiently performed in parallel. Therefore an iterative equation solver was used instead of the direct solver in the original sequential program.

Implementation. After parsing the command line, the ocean application sets various global parameters, e.g. the number of processors, the size of the ocean grid, the distance between individual grid points, as well as the error tolerance. Based on these values, the shared arrays are initialized: The psi1, psi3, psim1, and psim3 arrays represent the mean stream function at the current and previous timesteps, respectively. The psium and psilm arrays represent the stream functions at the interface between both layers, in the middle of the upper layer, and the middle of the lower layer, respectively. The ga, oldga, and gb, oldgb arrays represent the γ_a, γ_b functions from the current and previous timesteps, respectively. The tauz array represents the vertical component of the wind-stress curl, while the f array represents the Coriolis parameter. The work1 through work7 and temp arrays are used as temporary arrays during setup of the differential equations. The q_multi and rhs_multi arrays are used as inputs to the multigrid solver.

After these and several local arrays have been initialized, the remaining processors are activated and the ocean is simulated for the specified number of timesteps. Once the simulation has been completed, these processors are deactivated, while the first processor gathers timing statistics. Before the first timestep begins, the psi, psim, psib, psium, psilm, tauz and f arrays are initialized and the integral of the psibi arrays is determined. A subsequent barrier synchronization ensures that all processors have finished this initialization before the actual simulation starts.

The simulation proceeds for the specified number of timesteps, each timestep consists of 10 steps to solve the equation system described in Section 4.2.4. Barrier synchronizations between the individual steps ensure that all processors have finished the current step before moving to the next step.

- The first step consists of six different computations: First, the ga and gb arrays are initialized. Second, the Laplacian of the psi1 array, i.e. Ψ_1, is computed and stored in the work1 array. Third, the Laplacian of the psi3 array, i.e. Ψ_3, is computed and stored in the work3 array. Fourth, the difference between the psi1 and psi3 arrays, i.e. $\Psi_1 - \Psi_3$ is computed and stored in the work2 array. Fifth, the Ψ_2 stream function is computed and stored in the work3 array. Last, the psi1 and psi3 arrays are saved to the temp1 and temp3 arrays.
- The second step consists of three different computations: First, the psim1 and psim3 arrays are copied to the psi1 and psi3 arrays, respectively. Second, the Laplacian of the psim1, psim3 array is computed and stored in the work7 arrays. Last, the work1 and work3 arrays are updated with the corresponding values from the f array. Afterwards these arrays contain $f + \nabla(\Psi 1)$ and $f + \nabla(\Psi_3)$, respectively.
- The third step consists of three different computations: First, the Jacobians of the work1, temp1 and work3, temp3 arrays is computed and stored in the work5 arrays. Afterwards these arrays contain $J(f + \nabla(\Psi 1), \Psi_1)$ and $J(f + \nabla(\Psi_3), \Psi_3)$, respectively. Second, the original values of the psim1, psim3

arrays are restored from the temp1, temp3 arrays. Last, the Laplacian of the work7 arrays is computed and stored in the work4 arrays.
- The fourth step consists of two different computations: First, the Jacobian of the work2, work3 arrays is computed and stored in the work6 array. Afterwards the work6 array contains $J(f + \nabla(\Psi 1 - \Psi_3), \Psi_2)$. Second, the Laplacian of the work4 arrays is computed and stored in the work7 arrays. Afterwards these arrays contain the three-fold Laplacian of the original psim arrays and represents the lateral friction terms.
- The fifth step uses the work5, work6, work7 arrays to compute the ga and gb arrays, i.e. γ_a, γ_b, according to the equations described above.
- The sixth step initializes the q_multi and rhs_multi arrays based on the ga array and solves the corresponding partial differential equation using an iterative Red-black, Gauss-Seidel multigrid solver. The solutions is stored in the ga array, which is saved to the oldga arrays afterwards. Recall that the oldga array is used to provide an initial guess during the next iteration.
- In the seventh step, each processor computes the integral of the local part of the ga array and updates the global psibi variable accordingly.
- The eighth step consists of two different computations: First, the ga array is updated according to the above equations. Second, the q_multi and rhs_multi arrays are initialized based on the gb array and the corresponding partial differential equation is solved using the same multigrid solver as above. The solution is stored in the gb array, which is saved to the oldgb arrays afterwards.
- In the ninth step, the solutions of the partial differential equations stored in the ga, gb arrays are used to update the work2 and work3 arrays.
- In the tenth step, the psi1 and psi3 arrays are updated from the work2 and work3 arrays to prepare for the next iteration.

Porting. Porting the ocean application described above was complicated by the size of the application and the large number of arrays. First of all, the PARMACS macros were either removed or substituted with calls to the corresponding routines from the shmem, emulation and performance counter libraries. Some data structures, e.g. barrier and lock structures, were removed or integrated into the other data structures.

All implicit accesses to remote memory were identified and replaced by explicit accesses using communication routines from the shmem and emulation libraries. The shared arrays were distributed as suggested for distributed shared-memory systems in the original sources, i.e. each processor allocates its local subgrid of the individual arrays. Since all processors are active at startup, some changes were made to the initialization routines to ensure that some code sections, e.g. I/O, are only executed by a single processor.

In the case of emulated multithreading, the data distribution is unchanged, but the work is distributed among the $p{\cdot}t$ threads on the p processors as if $p \cdot t$ processors were present. Apart from thread initialization, this is the only major change required to support emulated multithreading.

4.2.5 The Barnes application

The barnes application solves the classical N-body problem: Given a set of N particles, where each particle is defined by its mass, position, and velocity, the evolution of the particle system under the influence of gravitational forces is computed. This requires discretizing the time period into small time steps, calculating the gravitational forces between all particles and updating the particle positions, velocities, and accelerations accordingly in each time step. Since the range of the gravitational force is infinite, each particle is influenced by all other particles. As the number of particle pairs is $\binom{N}{2}$, a straight-forward algorithm requires $O(N^2)$ time for each time step. The barnes application uses the barnes-hut algorithm to achieve an $O(N \log N)$ time bound.

Algorithm. The barnes-hut algorithm [BH86] uses a hierarchical method based on constructing a tree of particles. All nodes in the tree represent a regular space cell of the particle system and the leaves contain at most one particle. The root node represents a space cell that is large enough to contain the whole particle system. Starting with the root node, the tree is constructed by subdividing the corresponding cell into up to eight subcells until the subcells contain no more than one particle. Note that the length of the subcells is one half of the length of the parent cell in all three dimensions.

The size of the root cell is determined in $O(N)$ time by examining the current positions of all particles. The tree is constructed by starting with the empty root cell and subsequently inserting all particles into the tree in arbitrary order. The insertion of a particle leads to subdivisions of tree nodes if there is more than one particle in the corresponding space cell. Empty subcells are not stored, i.e. the tree is adaptive. The expected runtime for the tree construction is $O(N \log N)$, since the insertion of a particle requires time proportional to the height of the tree, which is expected to be $O(\log N)$. Therefore the overall runtime for tree construction is $O(N \log N)$ as well.

For each internal node, the center of masses for all particles in any of the corresponding subcells is calculated. These values are used during the third step to approximate the interaction with the particles contained in these subcells. A traversal of the tree in reverse direction, i.e. starting with the leaves, is used to calculate the center of masses for all leaves and to propagate the corresponding information to all nodes in the tree. Since the expected height of the tree is $O(\log N)$, the number of nodes in the tree is $O(N)$ on average, hence the second step has an average runtime of $O(N)$.

The gravitational forces on all particles are computed in the third step. For each particle, the tree is traversed to compute the gravitational forces from all other particles that affect the current particle. The traversal starts at the root node and is governed by the following rules: If the center of masses in one of the subcells of the current node is well-separated from the current particle, the gravitational force caused by the particles in that subcell is approximated by the corresponding center of masses. Otherwise the traversal

continues recursively into the subcell. The dimensions of the current cell, the distance between the current particle and the center of masses of the subcell as well as a threshold is used to determine whether a given particle and subcell are well-separated. The threshold is usually user-defined and determines the accuracy of the approximation along with the duration of the individual timesteps and the accuracy of the mathematical operations. Since the tree is traversed for each particle during this step, the force calculation for each step requires $O(\log N)$ time, hence the third step has an overall runtime of $O(N \log N)$.

Implementation. After parsing the command-line, the barnes application sets various global parameters, e.g. the number of particles, the duration of a timestep, the number of timesteps, and the number of processors. Based on these values, the shared arrays are initialized as follows: The btab array is a shared array that holds the N particles. The mass, position, and velocity of the individual particles are initialized based on a Plummer model.

The ctab and ltab arrays are allocated for each processor and hold internal nodes and leaves of the particle tree, respectively. The size s_1, s_2 of these arrays is the number of particles times the fraction of leaves per particle and the fraction of nodes per particle, respectively, divided by the number of processors p:

$$s_1 = n \cdot f_{\text{leaves}}/p$$
$$s_2 = n \cdot f_{\text{nodes}}/p$$

The two fractions can be specified via the command-line, default values are 0.5 and 2.0, respectively.

The three local mybody, myleaf, and mycell arrays are used to hold pointers to elements in the shared btab, ltab, and ctab arrays, respectively. These arrays are required since the distribution of particles, leaves, and nodes to processors is changed in every timestep. The mybody array is initialized such that processor i holds pointers to n/p particles starting from the $i \cdot (n/p)$th particle. After initializing the local and shared arrays, the remaining processors are activated and the evolution of the particle system is determined for the specified number of timesteps. Once the simulation has been completed, these processors are deactivated, while the first processor gathers timing statistics.

In each timestep, the three phases of the barnes-hut algorithm described above as well as an additional load-balancing step are executed:

- The first processor creates a tree with an empty root node, a subsequent synchronization barrier ensures that all processors start building the particle tree at the same time. Each processor loads the particles from its mybodytab array into the tree as described above. Note that synchronization is required to ensure atomic updates of the tree. Nodes and leaves that are created during insertion of a particle are stored in the ctab and ltab arrays of the processor that inserted the particle.

- The particles are redistributed across the processors using a work-partition scheme, as the amount of work required to calculate the gravitational forces acting upon a particle is non-uniform. The number of interactions with other particles is used as a simple cost measure to reflect the amount of work associated with that particle. Note that the number of interactions is not known before the forces are calculated and changes dynamically across timesteps. However, provided that the time steps are small enough, the particle system evolves slowly between timesteps. Hence the number of interactions in the previous timestep is an useful approximation to the number of interactions in the current timestep.
  Based on this cost measure, the particles are distributed as follows: First of all the average cost per processor is calculated as the total cost of all particles divided by the number of processors p:

$$C_{\text{avg}} = C_{\text{tot}}/p$$

  Note that the total cost of all particles is equal to the cost of the root cell. The total cost is distributed across the processors such that a partition of size C_{avg} is assigned to each processor. The minimum and maximum cost for processor i, i.e. the start and end points of the corresponding partition, is calculated as follows:

$$C_{\text{min}} = C_{\text{avg}} \cdot i$$
$$C_{\text{max}} = C_{\text{avg}} \cdot (i+1)$$

  Based on the minimum and maximum costs per processor, all processors traverse the particle tree, summing the cost of all particles encountered. Once the sum is larger than the minimum cost, the encountered particles are allocated to the processor, until the sum exceeds the maximum cost for this processor. Note that the particles themselves are not distributed, only the mybodytab array that stores the pointer to the local particles is updated. No synchronization is required during the work distribution phase, , as the minimum and maximum cost values as well as the identical traversal of the particle tree ensure that the particle distribution is disjoint.
- During the third step, each processor computes the gravitational forces for its local particles. For each particle, the particle tree is traversed as described above, calculating the number of particle-particle and particle-cell interactions along the way. In addition, the cost of the particle, i.e. the number of interactions, is calculated.
- After calculating the gravitational forces, each processor updates the position, velocity, and acceleration of all its local particles as described above. In addition, the minimum and maximum positions in each dimension is determined. If any of the three local minimum or maximum dimensions exceeds the corresponding global dimensions, the global dimensions are updated accordingly. Note that synchronization is required to ensure proper

updates of the global dimensions. These dimensions are used during the next timestep to create the root node of the particle tree.

Porting. Porting the barnes application described above was complicated by the size of the application and the dynamic assignment of particles to processors. First of all, the PARMACS macros were either removed or substituted with calls to the corresponding routines from the shmem, emulation, and performance counter libraries. Some data structures were removed or integrated into the other data structures.

All implicit accesses to remote memory were identified and replaced by explicit accesses using communication routines from the shmem and emulation libraries. However, as the assignment of particles, nodes, and leaves to processors is dynamic, the target processing element of the remote access has to be stored explicitly for all pointers that are potentially involved in remote memory accesses. Due to the size of the barnes application, the identification of remote accesses and corresponding pointers was quite complex.

The six shared arrays were distributed across the processors as suggested in the original source code: The body array is distributed such that every processor i holds a subarray that contains n/p particles starting with the $i \cdot (n/p)$th particle. Note that the body array is still shared, i.e. the arrays on all processors start at the same address. The ctab and btab arrays were distributed as shared arrays to the corresponding processors. The mybodytab, mycelltab, and myleaftab arrays were distributed in the same way.

In the case of emulated multithreading, the data distribution is unchanged, but the work is distributed among the $p{\cdot}t$ threads on the p processors as if $p \cdot t$ processors were present. Apart from thread initialization, this is the only major change required to support emulated multithreading. Since all processors are active at startup, some changes were made to the initialization routines to ensure that some code sections, e.g. I/O, are only executed by a single processor.

4.2.6 The FMM application

Like the barnes application described in Section 4.2.5, the fmm application solves the classical N-body problem. In contrast to the previous approach, the fmm application is restricted to the two-dimensional case and uses the fast multipole algorithm. Note that the algorithm itself is not restricted to two dimensions, but the three-dimensional formulation of the algorithm is considerably more complex. The primary difference between the two algorithms is the calculation of cell-cell interactions in the fast multipole algorithm, while the barnes-hut algorithm calculates only particle-particle and particle-cell interactions. Another major difference between the two algorithms is the definition of well-separatedness and the use of multipole expansions in the fast multipole algorithm compared to centers of masses in the barnes-hut algorithm. The next section describes the fast multipole algorithm in detail.

Algorithm. Similar to the barnes-hut algorithm, the fast multipole algorithm uses a hierarchical method based on constructing a tree of particles. All nodes in the particle tree represent a regular space cell of the particle system, the root node represents a space cell that is large enough to contain all particles in the system. Note that each leaf in the particle tree can contain several particles instead of the single particle used in the barnes-hut algorithm.

Starting with the root node, the tree is constructed by subdividing each cell into four quarter-sized subcells until the corresponding leaf cell contains no more than the maximum number of particles per leaf:

- The size of the root cell is determined in $O(N)$ time by inspecting the current positions of all particles. The tree is constructed in the same way as in the barnes-hut algorithm. The only difference is the maximum number of particles per leaf: In the fast multipole algorithm, a leaf is only subdivided if the number of particles in the leaf exceeds the maximum number. Setting the maximum number of particles to one yields the barnes-hut tree-construction algorithm.
- Each cell is approximated by a linear-order series expansion of particle properties around the center of the cell, the so-called multipole expansion. The number of terms used in the expansion determines the accuracy of the interpretation, an infinite number of terms would yield the exact result. Note that the accuracy of the computation in the fast multipole algorithm is controlled by the number of terms in the expansions compared to the choice of well-separatedness as in the barnes-hut algorithm. In addition, the accuracy of both algorithms is determined by the length of the individual timesteps as well as the accuracy of the floating-point operations. The multipole expansions are determined in an upward pass of the particle tree, propagating the multipole expansions from the particles in the leaves to the root node.
- Before calculating the gravitational forces, each node divides all nodes whose corresponding cells are not well-separated from the parent cell into several lists. Two nodes a, b are said to be well-separated if the distance between the corresponding cells is larger or equal than the length of b. Each of the four lists contains cells that bear a special relationship to the current cell:
  - The U list of a leaf l contains all leafs that are adjacent to the leaf l.
  - The V list of a node c contains all well-separated siblings.
  - The W list of a leaf l contains all descendants of l's colleagues whose parents are adjacent to l, but which are not themselves adjacent to l.
  - The X list of a node c contains all nodes that have c in their W list.

  After constructing these lists for all nodes and leaves in the particle tree, the interaction of each node with all nodes in the corresponding lists is computed. The list construction ensures that no interactions are computed

with cells that are well-separated from the parent cell. Note that the interactions with nodes from different lists are different, details can be found in [SHHG93].
Internal nodes compute only interactions with nodes in their V and X lists and store the results as a local expansion series. The local expansion represents the interactions with all well-separated nodes. In addition, leaf nodes compute interactions with the nodes in their U and W lists and update the particles in the leaf accordingly. Note that the interactions between the particles in the leaf node and all nodes in the U list are not approximated in any way, while interactions with nodes in the W list are approximated by the corresponding multipole expansion
After the interaction lists have been computed for all nodes, the interactions of well-separated nodes on the particles in a leaf cell are represented by the local expansions in the ancestor nodes. These local expansions are propagated to the leaves by a downward pass of the particle tree. The particles in the leaf cells are subsequently updated according to the local expansion. The complexity of the force calculation phase is $O(nh^2m)$ where m is the number of terms used in the multipole expansion, n is the number of particles in the tree, and h is the number of levels in the tree [Sin93].

Implementation. The fmm application uses command-line arguments to pass several important parameters such as the number of particles, the type of the particle distribution, the number of processors, the number of terms in the multipole expansion, as well as the number and duration of timesteps. After parsing the command-line, these parameters are used to create the particle distribution. The particles themselves are stored in a shared array, an additional array of the same size stores pointers to the individual particles. The latter array is initialized such that the ith entry contains a pointer to the ith particle. In addition, several static arrays that are used during multipole expansion are initialized. After initialization is completed, the remaining processors are activated, such that the evolution of the particle system is simulated by all p processors. Once the simulation has been completed, these processors are deactivated, while the first processors gathers timing statistics.

At startup, each processor allocates and initializes the local particle array, which contains pointers to particles. The size of the array is the product of the total number of particles and a constant particle distribution factor, divided by the number of processors. The particle distribution factor is used to account for imbalances in the particle distribution. The local particle arrays are initialized such that every processor holds an equal number of particles. In addition, each processor allocates an array of internal nodes used to construct the tree. The number a of these nodes is given by

$$a = 4/3 \cdot \frac{\text{tol} \cdot \text{bdf} \cdot n \cdot p}{\text{occupancy} \cdot \text{max\_particles\_per\_node}}$$

Note that the box distribution factor bdf and the tolerance tol are used to account for imbalances in the distribution of cells to processors, while

the maximum number of particles per node times the occupancy gives the average number of particles per node.

A subsequent barrier synchronization ensures that all processors have completed initialization before the evolution of the particle system is simulated. The simulation consists of a single loop that is iterated for the specified number of timesteps. For each timestep, the simulation is performed in six different steps:

- In the first step, the particle tree is constructed. This step uses a different algorithm to construct the particle tree that significantly reduces the amount of synchronization required. First of all the size of the root cell is determined: Each processor calculates the minimum and maximum particle positions in both dimensions by inspecting all particles that belong to it. Afterwards the global minimum and maximum particle positions in both dimensions are determined by merging the local minimum and maximum positions. A barrier synchronization ensures that all processors have calculated their local minimum and maximum particle positions prior to the merge.
  After the size of the root cell has been determined, each processor initializes the internal cell and particle lists, i.e. destroys the previous particle tree. The particle tree is constructed by building local particle trees on each processor and merging these local trees into a global particle tree afterwards. Each processor constructs its local particle tree as described above by starting with an empty root node and subsequently inserting its local particles into the tree. Note that the root node has identical dimensions in all local particle trees, therefore two nodes in different local trees represent the same subspaces. This property simplifies the merging algorithm, which is described in the next paragraph.
  The algorithm used to merge a local particle tree into the global tree starts at the root nodes of both trees. Based on the type of these nodes, there are six different cases:
  – If the global node is empty while the local node is internal, the global node is substituted by the local node.
  – If the global node is empty while the local node is a leaf, the global node is substituted by the local node.
  – If both nodes are internal, all children of the local node are merged with their counterparts in the global tree by calling the merging algorithm recursively for each child.
  – If the global node is internal while the local node is a leaf, the local node is subdivided such that the local node becomes internal and the previous case applies.
  – If both nodes are leaves, the global node is removed from the tree and the particles in the global node are inserted in the local subtree starting with the local node. Afterwards the local node is inserted into the global tree.

  - If the global node is a leaf while the local node is internal, the global node is removed from the tree and the particles in the global node are inserted in the local subtree starting with the local node. Afterwards the local node is inserted into the global tree afterwards.

  Note that the local node is never empty, hence there are only six instead of nine different cases. This algorithm significantly reduces the amount of synchronization, as whole subtrees instead of particles are inserted in a single locking operation. In the barnes application, synchronization is a bottleneck, especially in the upper levels of the tree.

  During the construction of the particle tree, each processor also constructs the partition lists. There is one partition list for the leaves in the particle tree and another one for each level of the particle tree. After the particle tree has been constructed, these particle lists contain pointers to all leaves or internal cells that belong to the processor. The partition lists are used during the construction of the U, V, W, and X lists described below as well as during the force calculation phase.
- The U, V, W, and X lists are constructed in two steps: First the lists of siblings and colleagues are constructed for each cell, afterwards the U, V, and W lists are constructed. Note that the X list is not computed explicitly, as it is the dual of the W list. The two steps are separated by a barrier synchronization to ensure that the siblings and colleagues of each cell have been determined before the U, V, and W lists are constructed.

  Given a node c, recall that all siblings of the node are all other nodes that have the same parent. Given a node c, recall that all colleagues of the node are its siblings as well as its cousins, i.e. children of siblings of the parent. The lists of siblings and colleagues for each node is constructed in a downward pass of the particle tree: Each processor uses its partition lists to update all nodes in all internal nodes of the tree as well as the leaves.

  The U, V, and W lists are updated in a similar way, although an upward pass of the particle tree is used to construct the lists. The individual lists are constructed for all cells according to the definitions given above.
- Similar to the barnes application, the nodes and leaves are redistributed to processors prior to force calculation. However, the cost of an internal node is represented by the number of nodes in the V list, while the cost of a leaf is represented by the sum of all cycle counts for interactions with all nodes in the U, V, W, and X lists. The cycle count for a given list is approximated by a simple function. This function is parameterized by the number of terms in the multipole expansion as well as the number of particles in the corresponding cell. Note that the partition lists are reconstructed during the repartitioning of the nodes and leaves. A subsequent barrier synchronization ensures that the partitioning is completed before the force calculation phase.
- The force calculation phase consists of four steps: First of all, the multipole expansions of all leaves are calculated and propagated to their ancestors

by an upward pass of the particle tree. Afterwards the U, V, W, and X interactions are computed in another upward pass of the particle tree. A subsequent barrier synchronization ensures that these interactions have been computed for all nodes before the local multipole expansions and updated particle positions are calculated.
The local multipole expansion is propagated from the root cell to the descendants in a downward pass of the particle tree and evaluated at the leaves. The particle positions are updated according to the interactions with other particles/cells. The different partitions of the particle tree are based on the partition lists that are maintained by each processor.
In the last step, each processor updates the list of its local particles by examining all leaves in the corresponding partition list and adding all particles in these leaves to its particle list. Note that the particle tree is reconstructed in every timestep, hence the need to update the particle lists.

Porting. Porting the fmm application was complicated by the size of the application and the dynamic partitioning of the particles. First of all, the PARMACS macros were either removed or substituted with routines from the shmem, emulation, and performance counter libraries. Some data structures could be removed, since the shmem and emulation libraries allow synchronizations on arbitrary longwords and barriers require no data structure. All implicit accesses to remote memory were identified and replaced by explicit accesses using routines from the shmem and emulation libraries. .

Due to the dynamic partitioning of the particle system, the target processing elements for remote memory accesses have to be stored explicitly for all pointers that are potentially involved in such accesses. As the fmm application is even larger and more complex than the barnes application, the identification of implicit remote accesses was quite complex. Unfortunately, the decision to store the address and the corresponding processing element in different locations created the potential for race conditions: The tree merging algorithm in the fmm benchmark allows simultaneous accesses to the individual nodes in the tree as long as only one processor updates the node. Therefore some of the read accesses will return the updated contents of the node, while other read accesses will return the original contents of the node. This is no problem, as the original contents are still valid, i.e. the corresponding nodes are still part of the tree, although in a different position. However, if the address and the number of the processor is stored in different locations, some read accesses will return the original address along with the updated processor number and vice versa.

This problem was detected during the evaluation of emulated multithreading on the Cray T3E as the problem manifests itself only with a large number of particles and multiple processors. Due to resource constraints, the fmm benchmark was tested with smaller problem sizes only during the porting phase. The problem can be solved in two different ways: All read and write accesses to the nodes in the tree are separated by locks. Due to the large

number of these accesses, this solution will probably have a significant impact on the tree merging performance. The second solution is to store the address and the processor number in the same memory location. This can be accomplished by storing the processor number in the unused upper part of the address and insert or extract both values before or after each access. Due to the large number of accesses to such locations, the second solutions requires some work.

The shared particle and particle list arrays are distributed across the processors such that every processor holds an equal number of array elements. The local cell and particle list arrays are allocated at the corresponding processor. In the case of emulated multithreading, the data distribution is unchanged, but the work is distributed among the $p \cdot t$ threads on the p processors in the same way, as if $p \cdot t$ processors were present. Apart from thread initialization, this is the only major change required to support emulated multithreading.

5. Evaluation : Compaq XP1000

As described in Chapter 2, emulated multithreading is designed to tolerate long latency events by using fine-grained multithreading and asynchronous communication. The implementation described in Chapter 3 is targeted at tolerating remote memory references in massively parallel processors, since such accesses cause long latencies and are identified easily. However, a basic understanding about the behavior of programs using emulated multithreading as well as the associated overhead can already be obtained from an evaluation on a single-processor workstation. The Compaq XP1000 workstation used in the single-processor evaluation of emulated multithreading is based on the 21264 implementation of the Alpha architecture. Compared to the 21164 processor used in the Cray T3E, the 21264 is better suited for emulated multithreading: The 21264 has significantly larger first-level caches and supports out-of-order execution and sophisticated branch prediction. Hence the evaluation on the Compaq XP1000 workstation and the Cray T3E should provide useful insights into the overhead associated with emulated multithreading on different implementations of the Alpha architecture. In addition, performing the evaluation on a local workstation is more flexible compared to an external system like the Cray T3E, since the workstation can be configured as needed. This chapter covers the evaluation of emulated multithreading on single-processor systems, while the evaluation of multithreading on massively parallel processors is described in Chapter 6.

Emulated multithreading is evaluated on the Compaq XP1000 workstation using the six benchmarks described in Chapter 4, i.e. fft, lu, radix, ocean, barnes and fmm. For each benchmark, several experiments are performed to determine the runtime of emulated multithreading across a wide range of thread numbers, problem sizes and code conversion options. These runtimes are subsequently compared with the runtimes obtained via single-threaded execution (base) as well as multithreaded execution using POSIX threads (posix).

Section 5.1 describes the architecture and software environment of the Compaq XP1000 workstation. The experimental methodology is covered in Section 5.2, while Section 5.3 describes the impact of the code conversion process on the size and structure of the assembler sources. Sections 5.4 to

5.9 describe the experimental results for the individual benchmarks, while Section 5.10 summarizes the results for all benchmarks.

5.1 Compaq XP1000

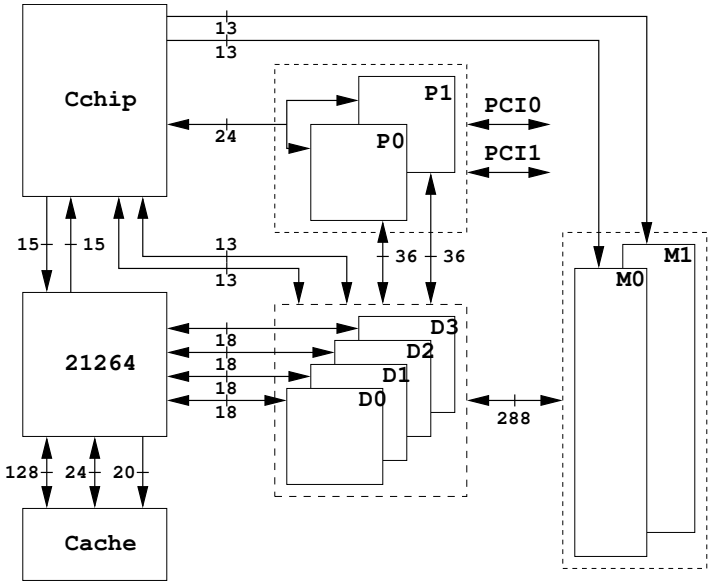
The Compaq XP1000 is a single-processor workstation based on the Alpha 21264 microprocessor and the 21272 chip set. The internal architecture of the system is illustrated in Figure 5.1. The information presented in this section was gathered from several sources, i.e. [Cor99][AXP00a][AXP99].

The 21264 microprocessor operates at 500 MHz and is connected to the 21272 chip set by the 15 bit system address and 72 bit (64 bit data) system data bus. These busses operate at 333 MHz, yielding a maximum bandwidth of 2.6 GB/s for the system data bus. The 4 MB second-level cache is connected by a dedicated 128 bit data bus, a 20 bit tag bus as well as a 24 bit address bus. These busses operate at 166 MHz, yielding a bandwidth of 2.6 GB/s for the cache data bus. The internal architecture of the 21264 microprocessor is described in Section 5.1.1.

The 21272 chip set consists of three different chips, e.g. Cchip, Dchip and Pchip, and can be used for one-, two-, or four-processor systems. The XP1000 workstation uses a combination of one Cchip, four Dchips and two Pchips. The Cchip is connected to the system address bus of the 21264 microprocessor by two uni-directional, clock-forwarded busses: One of the busses is used to issue commands and addresses from the processor to the Cchip, while the other is used to transfer results and addresses from the Cchip to the processor. Both busses are 15 bit (13 bit data) wide and operate at 333 MHz. However, it takes at least four cycles to transfer one complete command or address across one of these busses. The Pchips are connected to the Cchip by a bidirectional bus that is shared between the two Pchips. This 24 bit bus operates at 83 MHz and is used to transfer commands and addresses from the Cchip to the Pchips and vice versa. Note that it takes at least two cycles to transfer a command or address across this bus. The four Dchips are connected to the Cchip by a shared bidirectional 13 bit wide bus operating at 83 MHz. The Cchip provides two identical copies of this bus, two Dchips share one of the copies. Finally, the Cchip controls the two memory arrays via an unidirectional 13 bit bus operating at 83 MHz. The architecture of the Cchip is covered in Section 5.1.2.

The four Dchips are connected to the system data bus of the 21264 microprocessor by a 72 bit (64 bit data) bidirectional, clock-forwarded bus operating at 333 MHz, yielding a bandwidth of 2.6 GB/s. Each Dchip is connected to a 18 bit (16 bit data) segment of the system data bus. The two memory arrays are connected to the Dchips by a 288 bit (256 bit data) bus operating at 83 MHz, yielding a bandwidth of 2.6 GB/s. Note that the bandwidth of the memory data bus is identical to the bandwidth of the system data bus. Each Dchip is connected to a 72 bit (64 bit data) segment of the memory

Fig. 5.1. Architecture of the Compaq XP1000 workstation



data bus. In addition, the Dchips are connected to the Pchip and the Cchip as described above. The architecture of the Dchip is covered in Section 5.1.3.

The Pchips are connected to the Dchips by two bidirectional 36 bit (32 bit data) busses operating at 83 MHz, yielding a bandwidth of 333 MB/s per bus. Each Dchip is connected to the two Pchips by a 9 bit (8 bit data) segment of both busses. The Pchips provide two PCI busses and are described in Section 5.1.4. Section 5.1.5 describes the memory system, while Section 5.1.6 describes the peripherals used in the XP1000 workstation.

5.1.1 Processor

The 21264 microprocessor is an implementation of the Alpha architecture, a 64 bit architecture developed by Digital Equipment Corporation. A detailed description of this architecture as well as all implementations can be found in Appendix A. The 21264 microprocessor is a super-scalar processor that can issue up to six instructions (four integer and two floating-point) in each cycle, although the sustained issue rate is limited to four instructions due to the fetch bandwidth. The instructions are issued out-of-order. i.e. as soon as their arguments are available. However, all instructions are retired in program order to ensure sequential program semantics. Register renaming is used to resolve name dependencies, i.e. the 21264 processor contains 80 integer and 72 floating-point registers that are mapped to the 32 integer and 32 floating-point registers defined by the Alpha architecture.

The four integer execution pipelines are organized in two clusters. One execution pipeline in each cluster supports simple integer instructions as well as integer and floating-point load and store instructions. In addition, one of the execution pipelines supports integer multiply instructions, while the other supports motional video instructions. All integer execution pipelines support 64 bit and 32 bit operands. The two floating-point execution pipelines support all arithmetic instructions on single- and double-precision floating-point operands as well as floating-point control instructions. Note that floating-point division is handled by a non-pipelined divider that is associated with one of the execution pipelines.

The 21264 microprocessor contains two first-level data and instruction caches and supports an external second-level cache. The data cache is a 64 KB two-way set associative cache with 64 byte cache-lines that uses a write-back, write-allocate protocol. The data cache is dual-ported and supports two independent accesses in each cycle. The instruction cache is a 64 KB two-way set-associative cache with 64 byte cache-lines. The external second-level cache is a direct-mapped cache ranging from 1 to 16 MB in size connected by a dedicated 128 bit data bus. The Compaq XP1000 uses a 4 MB second-level cache that operates at 166 MHz, yielding a bandwidth of 2.6 GB/s between the cache and the 21264 microprocessor. Note that each integer load that hits in the first-level cache incurs a latency of three cycles, while each floating-point load that hits in the first-level cache incurs a latency of four cycles. Loads that miss the first-level caches, but hit in the second-level cache, incur a latency of 12 cycles or more. In order to increase performance of load and store instructions, the 21264 microprocessor contains a miss address file that buffers and merges outstanding loads as well as a write buffer that buffers and merges stores.

The external data bus is 72 bit (64 bit data) wide and operates at 333 MHz, yielding a bandwidth of 2.6 GB/s. Note that the 21264 uses two unidirectional address and command busses instead of a single bidirectional bus, one for each direction. The Alpha architecture defines a 64 bit virtual address space, implementations are required to support at least 43 bit virtual address space as long as the unused bits are checked to be zero. The 21264 microprocessor implements a 48 bit virtual address space as well as a 44 bit physical address space. Detailed information about the internal architecture of the 21264 microprocessor can be found in Section A.3.9.

5.1.2 Cchip

The Cchip controls the memory and I/O subsystems by receiving requests from the processor and the Pchips and issuing commands to the Dchips and Pchips. The actual data transfers are performed by the Dchips and Pchips, i.e. the Cchip does not perform any data transfers. The Cchip provides two or four independent system address ports that connect to the individual processors. Each system address port consists of two unidirectional, clock-forwarded

busses: One to transfer command and addresses from the Cchip to the processor, the other to transfer commands and addresses from the processor to the Cchip. Note that it takes up to four cycles to transfer a single command across one of these busses. Each bus is 15 bit wide and operates at 333 MHz.

The Cchip provides four memory command and address ports, i.e. can control up to four independent banks of synchronous DRAM. Each of these ports consists of 15 address and 8 control signals and operates at 83 MHz. The memory command and address ports are only used to issue memory requests, the actual data transfers are handled by the Dchips. The Cchip provides a command and address port that is used to control up to two Pchips. The port is shared between the Pchips and consists of 24 address and 11 control signals and operates at 83 MHz. However, it takes at least two cycles to transfer a single command across these busses.

The Cchip provides an unidirectional bus to control up to eight Dchips. There are two identical copies of this bus, each one supports up to four Dchips. Each control bus is 13 bit wide and operates at 83 MHz. Finally, the Cchip provides an 8bit bus to handle external interrupts, flash memory and other auxiliary devices.

The internal architecture of the Cchip is based on a central arbiter as well as four request queues, one for each memory array. Requests arriving from the processors and Pchips are placed in a central dispatch register before they are forwarded to the appropriate request queue. Even requests that do not access the memory arrays are placed in the request queues for the memory arrays. If a request cannot be placed in the dispatch register, it is temporarily held in so-called skid buffers. The arbitration logic selects requests from the memory request queues and issues up to three commands per request to the processor, memory, Dchips, or Pchips. In addition, the arbitration logic ensures that ordering requirements for the individual requests are maintained.

5.1.3 Dchip

The Dchips are responsible for the actual data transfers initiated by the Cchip. The Dchip provides two memory data ports that connect the Dchip to the memory arrays. Each port is 36 bit (32 bit data) wide and operates at 83MHz. In addition, the ports can be configured as a single 72 bit (64 bit data) port as well as two 18 bit (16 bit data) ports. The Compaq XP1000 uses four Dchips to interface the 288 bit (256 bit data) wide memory arrays, i.e. has to use the single port configuration. The Dchip provides four bidirectional, clock-forwarded processor data ports, one for each processor. Each port is 9 bit (8 bit data) wide and operates at 333 MHz, yielding a bandwidth of 333 MB/s. The ports can be configured as a single 36 bit (32 bit data) or two 18 bit (16 bit data) wide ports. The Compaq XP1000 is a single processor system and contains four Dchips that interface to the 64 bit system data bus, hence the processor data ports on the Dchips are configured as two 18 bit ports, although only one port is used.

The Dchip provides two bidirectional data ports for the Pchips, each port is 9 bit (8 bit data) wide and operates at 83 MHz. Depending on the number of Dchips and Pchips in a system, these ports can be configured as a single 18 bit (16 bit data) port or two multiplexed half-byte ports. The Compaq XP1000 uses four Dchips and two Pchips, hence the ports are configured a two single-byte ports. As described above, the Dchip is controlled from the Cchip by a bidirectional control bus.

The internal architecture of the Dchip is based on a crossbar switch that connects the individual data ports. The Dchip supports transfers between the processors, Pchips and memory arrays as well as between the four processor ports and the two Pchip ports themselves. Note that transfers between the two memory arrays are not supported. In addition, the Dchip provides several queues that enable buffering on the processor, memory, and Pchip ports. The data is not interpreted in any way, i.e. there is no kind of error checking and reporting.

5.1.4 Pchip

The Pchip provides a 64 bit PCI interface operating at 33 MHz. The Pchip is controlled by the Cchip via the Cchip command and address port, a bidirectional 24 bit port operating at 83 MHz. The data from and to the PCI interface is transferred to and from the Dchips via a 36 bit (32 bit data) bidirectional bus operating at 83 MHz. This bus can be configured as a 36 bit bus that transfers four bytes per cycle or a 40 bit bus that transfers eight nibbles per cycle. The latter configuration is used in systems with eight Dchips.

The internal architecture of the Pchip is based on a central PCI bus arbiter and several queues that decouple the PCI bus from the Dchip data bus and the Cchip address bus. There are two separate queues for each direction, one for data and one for addresses, i.e. a total of four queues.

5.1.5 Memory

The Compaq XP1000 workstation supports two independent memory arrays, each array is controlled by one of the memory control ports of the Cchip. Each memory arrays consists of four 72 bit (64 bit data) memory modules and provides a 288 bit (256 bit data) data bus. Each of the four Dchips is connected to a 72 bit segment of the memory data bus as described above. The individual memory modules are industry-standard PC100 synchronous DRAM operating at 83 MHz.

Although all four modules in the array have to be populated, it is possible to populate only one of the two memory arrays. Interleaving increases performance if both arrays are populated. The Compaq XP1000 supports between 128 MB (four 32 MB modules) and 2 GB (eight 256 MB modules) of memory. The system used during the evaluation of emulated multithreading

was configured with 640 MB of memory, i.e. 512 in one array and 128 MB the other array.

5.1.6 Peripherals

As described above, each of the two Pchips in the Compaq XP1000 provides a 64 bit PCI bus operating at 33 MHz. One of the PCI busses is configured as a 32 bit bus and provides two 32 bit PCI slots as well as a PCI-PCI bridge that is used to connect a secondary PCI bus to the primary PCI bus. The secondary PCI bus contains the embedded Ethernet and SCSI controllers. The Intel 21143 Ethernet controller provides a full-duplex 10/100 Mb Ethernet interface, while the Qlogic 1040 SCSI controller provides an ultra-wide SCSI interface.

The other PCI bus is configured as a 64 bit bus and provides two 64 bit PCI slots as well as one 32 bit PCI slot. Apart from the PCI slots, the PCI bus contains the Cypress 82C693 multi-function controller that provides two EIDE ports, two USB ports, as well as keyboard and mouse ports. In addition, the controller contains a PCI-ISA bridge, a realtime clock, as well as an interrupt controller. The PCI-ISA bridge provides a 16 bit ISA bus that is connected to a 16 bit ISA slot as well as multi-I/O and sound chips. The multi-I/O chip provides a floppy drive interface, a parallel and two serial ports, while the ESS1887 sound chip provides support for simultaneous playback and record of 16 bit audio data.

The Compaq XP1000 workstation used during the evaluation of multithreading is configured with a 4.3 GB Seagate Cheetah hard disk, a 32-speed CD-ROM drive, as well as a standard floppy drive. The hard disk is connected to the embedded SCSI controller, the CD-ROM drive is connected to one of the two EIDE ports in the multi-function controller, and the floppy drive is connected to the corresponding interface provided by the multi-I/O chip. Graphics is provided via an Elsa Gloria Synergy card based on the 3Dlabs Permedia2 chip and 8 MB of SGRAM. The PCI graphics controller uses one of the 64 bit slots on the first PCI bus.

5.1.7 Software Environment

The Compaq XP1000 workstation runs under the Tru64 UNIX operating system formerly called Digital Unix. The Tru64 operating system is a 64 bit system based on the Mach kernel and is compliant with the UNIX98 and System V Release 4 standards. A detailed description of the Tru64 operating system is outside the scope of this chapter, corresponding information can be found in [Tru01].

The development environment is provided by the Tru64 Developers Toolkit. The toolkit contains a C compiler, debugger, profiling and analysis tools as well as the Spike post-link optimizer. The C compiler is an ANSI-C

compliant implementation of the C language that produces highly optimized code. The ladebug debugger supports source-level debugging of Ada, C, C++, and Fortran languages as well as debugging of multithreaded applications. The profiling and analysis tools consist of the third degree memory profiling, pixie basic block profiling, and hiprof execution profiling tools. These tools are based on the ATOM framework [SE94] which can be used to implement custom-specific analysis tools. The visual threads tool supports the analysis of multithreaded applications. Finally, the Spike post-link optimizer supports whole-program optimizations and is based on the OM framework [SW93]. The Digital Continuous Profiling Infrastructure (DCPI) [ABD+97] permits continuous profiling of all processes including the kernel with low overhead. The tool is based on statistical sampling of the performance counters in the 21264 microprocessor. Unfortunately, the Compaq XP1000 workstation used during the evaluation of emulated multithreading still contains the original 21264 as opposed to the newer 21264A microprocessor. The latter implementation supports ProfileMe counters [DHW+97] that can be used to gather a complete execution profile of selected instructions instead of the summary statistics provided by traditional performance counters.

The Compaq XP1000 workstation used during the evaluation of emulated multithreading runs under version 5.0A of the Tru64 operating system, although the operating system was subsequently updated by installing the second aggregate patch cluster. The workstation is configured as a standalone system and stripped down to the essential system services: Apart from the ssh daemon, all non-essential daemons and processes were deactivated. The installed version of the development tools consists of Compaq C version 6.4 , Visual Threads version 2.0, Ladebug version 4.0, Graphical Program Analysis Tools version 3.1, Program Analysis Tools version 2.0, and Spike version 5.1. In addition, the DCPI version 3.9.2.2 toolkit was used to gather profiling information via the performance counters.

5.2 Methodology

All three versions of the benchmarks, i.e. base, posix and emulated multithreading, are based on the original sources as distributed in the SPLASH2 benchmark suite. For each benchmark, the corresponding source and header files are processed by the m4 macro processor [Sei00] in order to replace the original PARMACS macros. The resulting source and header files are subsequently processed by the compiler. The compiler uses the same set of optimization flags for all versions of the benchmarks: -fast and -arch=ev6. The former flag enables a collection of optimizations by assuming ANSI-compliant aliasing, enabling intrinsics, reordering of floating-point instructions, inter-file optimization, and faster math routines. In addition, the implied optimization level enables local optimizations, recognition of common subexpressions, inline expansion of static and global procedures, as well as global optimizations

like code motion, strength reduction, test replacement, split lifetime analysis, loop unrolling, code replication, and scheduling. The latter flag enables the use of instructions that belong to one of the Alpha architecture extensions supported by the 21264 microprocessor. In addition, the code scheduling phase produces code that is tuned to the 21264.

The original sources were modified such that the cycle counter is used for timing measurements. Since the cycle counter is incremented in each clock cycle, using this cycle counter enables very accurate timing measurements with high resolution. Access to the cycle counter is provided by the performance counter library (PCL) [BM98], i.e. the PARMACS macros used for timing measurements are substituted by calls to the corresponding routines of this library. The results of the timing measurements are rather large integers and have to be represented by 64 bit integers, hence the formatting used during the output of the timing results had to be adapted as well.

In the case of emulated multithreading, the assembler source produced by the compiler is processed by the assembler converter as described in Section 3.5. The project-specific configuration files are created manually as the high-level language converter has not been implemented yet. The converted assembler sources are subsequently passed to the assembler. The assembler sources are explicitly created for all versions of the benchmarks, even for the base and posix versions. Some compilers perform additional optimizations, e.g. substitution of static arrays by dynamic arrays created at runtime, if they are allowed to create the executable directly.

For the base version, all invocations of the PARMACS macros are replaced during m4 macro processing in the following way: All declarations, e.g of locks and barriers, are replaced by integer variables, the other macros are replaced by empty statements. The resulting code can only be used on single-processor machines as all parallelization constructs have been effectively removed.

The posix version uses an implementation of the PARMACS macros based on POSIX threads [ANMB97][AMBN98]. This implementation was provided by Ernest Artiaga from the Universitat Polytecnica de Catalunya (UPC) in Barcelona, Spain. All invocations of the PARMACS macros in the original sources are replaced during m4 macro processing by calls to the corresponding routines in the provided library. Note that the implementation used during the evaluation of emulated multithreading is configured to use advanced locks instead of the simpler spin locks.

For the versions using emulated multithreading, the invocations of the PARMACS macros are replaced by calls to the corresponding routines of the emulation library: The BARRIER() macro is substituted by a call to the EMUthread_barrier() routine, the LOCK() and ALOCK() macros are substituted by macros based on the EMUthread_cswap() routine. All declarations are replaced by integer types, the GMALLOC() array is replaced by a call to the standard malloc() routine followed by memset() such that the allocated memory blocks are initialized immediately. This is necessary as the SPLASH2

benchmarks frequently assume that GMALLOC() returns an initialized block of memory and therefore fail to initialize such arrays properly.

Several experiments are performed for each benchmark, using a wide range of configurations. The runtimes of the base and posix versions are compared to the runtime of several versions using emulated multithreading. The individual versions are created by restricting the size of basic blocks to 4, 16, or 64 instructions (g004, g016, g064), using basic blocks (bblk), as well as enabling super block optimization (sblk). For each benchmark, the base version as well as the six multithreaded versions are executed with identical parameters using three different problem sizes: the default problem size as defined in the SPLASH2 benchmark report as well as two and four times that size.

For each problem size, the multithreaded versions are executed with 1,2,4,8, and 16 threads in order to determine the impact of the number of threads. Altogether,

$$3 \times (6 \times (1 + 5 \times (5 + 1))) = 558$$

experiments were performed, 93 for each benchmark. All runtimes reported by the benchmarks were automatically extracted from the corresponding output and reassembled such that there is one file for each combination of benchmark, version, and problem size, that contains the corresponding runtime for all numbers of threads. The automatic extraction is based on a combination of shell [Bli96] and awk [DR97] scripts that were extensively tested in order to ensure the accuracy of the results. There are two different versions of each file: one version contains the runtime measurements as a number of clock cycles, the other contains the runtimes in seconds. Throughout this chapter, only the latter version of the files is used since the corresponding values are more intuitive.

All data files are formatted to allow a direct import into the xmgrace application that was used to create the corresponding figures. The hotlink feature of the xmgrace application was used to link the individual figures to the corresponding data files, i.e. updates to the data files are automatically reflected in the figures. The figures used to illustrate the results of the individual benchmarks are all structured in the same way: The horizontal axis reflects the number of threads, while the vertical axis represents the runtime in seconds. In each figure, seven curves are used to illustrate the results: The circles represent the runtime of the base version, while the squares, diamonds, upward triangles, leftward triangles, upward triangles and rightward triangles represent the runtime of the g004, g016, g064, bblk, sblk, and posix versions, respectively. Note that a vertical baseline is provided in order to make the base results better identifiable. Three of these figures are provided for each of the six benchmarks, one for each problem size.

5.3 Code Conversion

This section characterizes the impact of the code conversion process on the size and structure of the assembler sources for all six benchmarks. The corresponding information is gathered from the statistics for the original and modified code provided by the assembler converter. Figures 5.2, 5.3, 5.4, 5.5, 5.6, and 5.7 illustrate the corresponding statistics for the fft, lu, radix, ocean, barnes, and fmm benchmarks, respectively. Note that these statistics only cover the internal procedures of each benchmark as all other procedures are not affected by the code conversion process.

All figures are structured in the same way and contain the statistics for the original and modified sources for each of the five different versions of a benchmark. For each version, the left bar represents the instruction mix for the original assembler source. Note that this is the same for all versions of a given benchmark, but is replicated to ease comparison between original and modified instruction mixes. Each bar consists of a stack of segments, the height of the individual segments represents the number of instructions from the corresponding instruction group.

There are ten different instruction groups: The int_memory group contains all instructions that load and store integer data types to and from memory as well as instructions used to initialize constants. The int_control group contains direct and indirect branches based on integer data types. The int_operate group contains all arithmetic instructions that operate on integer data types, while the int_logic group contains logical and shift instructions and conditional moves. The int_byte group contains all instructions defined in the byte and word extension to the Alpha architecture as described in Section A.2.3, with the exception of the corresponding load and store instructions. The int_media group contains all instructions defined in the motional video extension to the Alpha architecture as defined in Section A.2.3. The int_misc group contains all integer instructions that do not belong to any of the other instruction groups. The fp_mem group contains all instructions that load and store floating-point data types to and from memory, similar to the int_mem group for integer data types. The fp_control group contains only conditional branches that operate on floating-point data types, as there are no unconditional or indirect branches of this kind. The fp_operate group contains all arithmetic instructions that use floating-point data types. The individual instructions are assigned to one of these instruction groups according to the information provided in the platform-specific configuration file.

Looking at the figures, it is evident that the total number of instructions in the modified assembler sources decreases with growing instruction block size in all cases: The g004, g016, and g064 versions restrict the instruction blocks to 4, 16, and 64 instructions respectively, while the bblk and sblk versions use basic and super blocks as instruction blocks. Compared to the bblk version, restricting the instruction block size to 64 instructions causes only a small increase in the number of instructions. This indicates that al-

Fig. 5.2. Original and Modified Instruction Mix for the FFT Benchmark

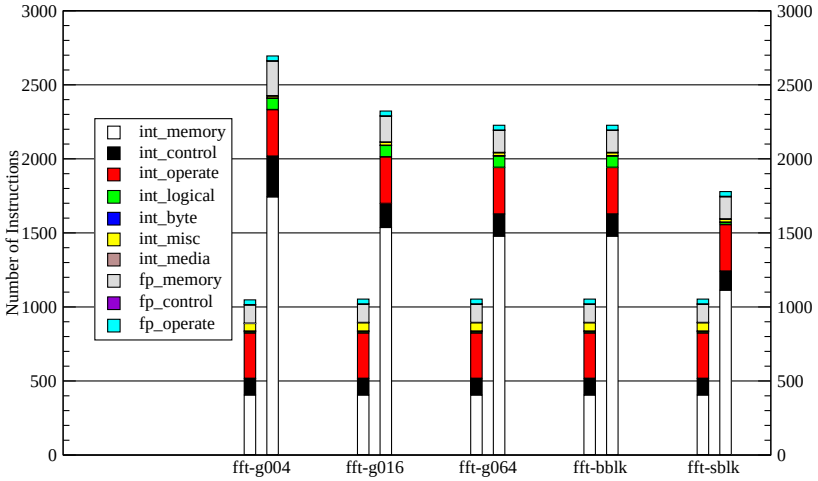
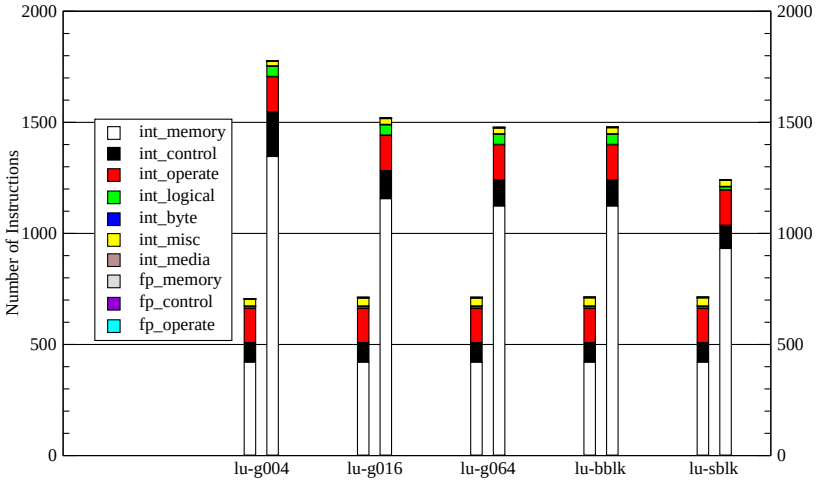


Fig. 5.3. Original and Modified Instruction Mix for the LU Benchmark



most all basic blocks are smaller than 64 instructions anyway. In the case of the lu benchmark, basic blocks seem to be no larger than 16 instructions, since the g016, g064 and bblk versions use an almost identical number of instructions. For all other benchmarks, using a restricted instruction block size of 16 instructions causes a slight increase in the total number of instructions compared to the bblk version. Using a restricted instruction block size

Fig. 5.4. Original and Modified Instruction Mix for the RADIX Benchmark

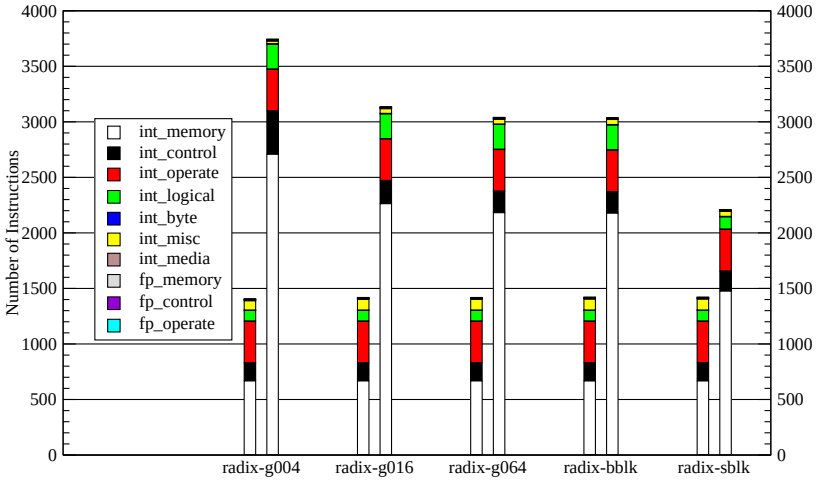
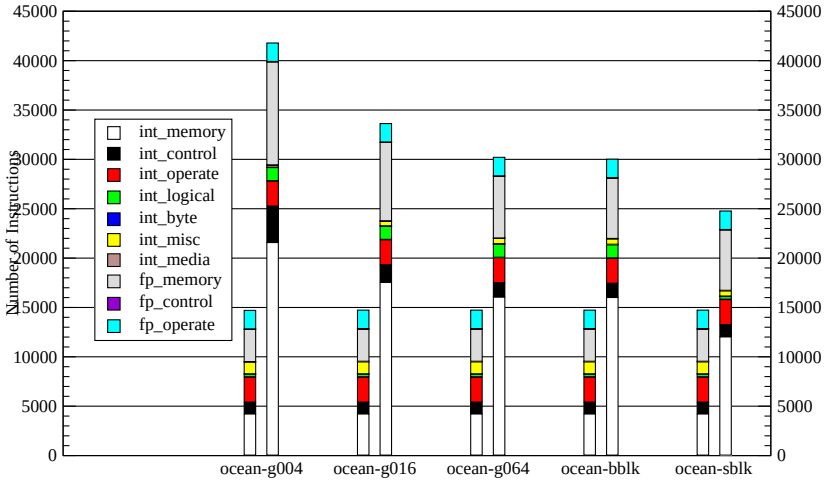


Fig. 5.5. Original and Modified Instruction Mix for the OCEAN Benchmark



of 4 instructions causes a noticeable increase in the number of instructions compared to the bblk version.

Restricting the size of the instruction blocks increases the number of instruction blocks as well as the total number of instructions, since context switch code has to be generated for each instruction block. The statistics for the converted assembler source support this argument: Compared to the bblk version, the g004, g016, and g064 versions of all benchmarks use more instruc-

Fig. 5.6. Original and Modified Instruction Mix for the BARNES Benchmark

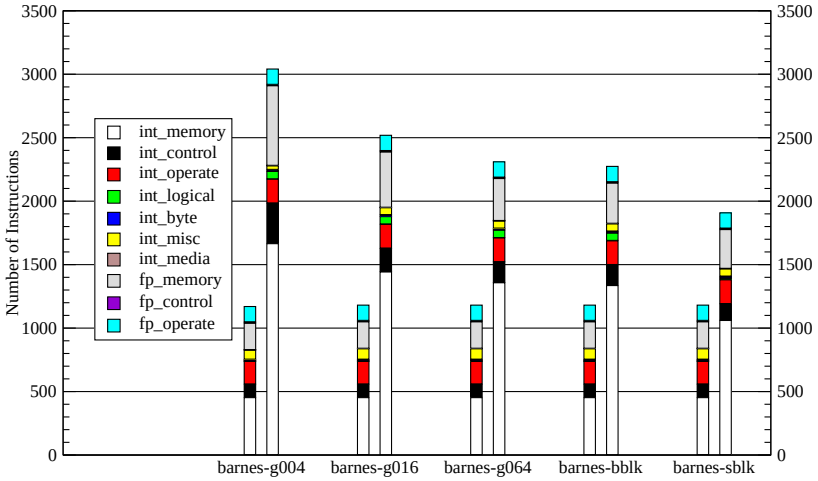
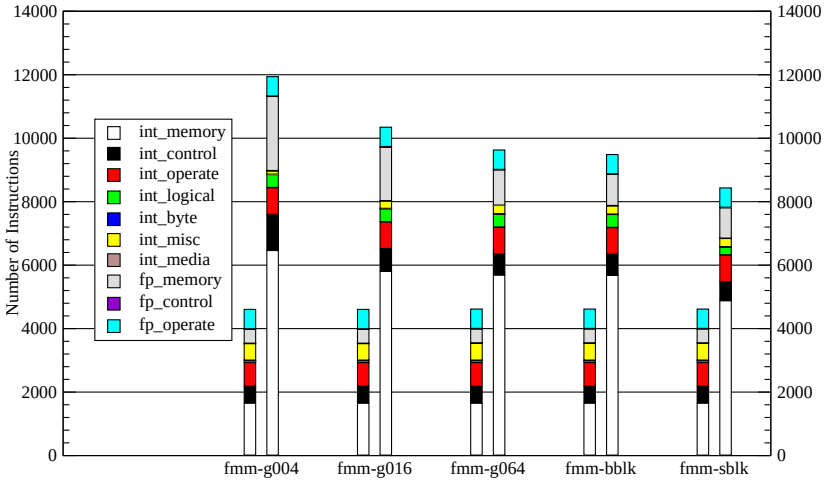


Fig. 5.7. Original and Modified Instruction Mix for the FMM Benchmark



tions from the int_mem, fp_mem, and int_control groups while the number of instructions from the other groups is not affected by restricting the instruction block size. The increase in the number of instructions from the int_mem and fp_mem groups, i.e. integer and floating-point load and store instructions, is caused by the increased number of save and restore operations due to the larger number of context switches. The increase in the number of instructions

from the int_control group is caused by the subroutine return instruction that is used at the end of each instruction block.

The super block optimization seems to be quite effective as it reduces the number of instructions noticeably: Compared to the bblk version, the sblk version uses less instructions from the int_mem, fp_mem, int_ctrl and int_logic groups. The number of instructions from the other groups is not affected by super block optimization. These changes are caused by merging multiple basic blocks into one super block: Since live ranges are allowed to cover several basic blocks as long as these blocks belong to the same super block, fewer save and restore operations are necessary, the number of load and store instructions is reduced: If the live range belongs to an integer register, the number of instructions from the int_mem group is reduced. If the live range belongs to a floating-point register, the number of instructions from the fp_mem group is reduced. However, additional save and restore operations may be needed to handle side entrances to the live range, if the live range covers more than one basic block. As the total number of instructions from the int_mem and fp_mem groups still decreases significantly, the latter effect is usually minor.

The reduced number of instructions from the int_control group is caused by the reduced number of instruction blocks, i.e. the return instruction at the end of each instruction block is placed at the end of each super block instead of each basic block. The instruction sequences used to calculate the target of a branch contain conditional moves which were assigned to the int_logic group. As long as the branch targets are inside the same super block, these instruction sequences can be replaced by a single branch instruction, hence the reduced number of instructions from the int_logic group.

Although the super block optimization is quite effective in reducing the number of instructions, the sblk versions of all six benchmarks still use up to twice as many instructions than the corresponding base versions, hence super block optimization should be improved further, e.g. by using profiling information. However, this applies only for the internal procedures, i.e. the increase in the total number of instructions for the whole program depends on the number and size of the internal procedures. The impact of the increased code size on the performance of the individual benchmarks is discussed in the following sections.

5.4 FFT

The project-specific configuration file for the fft benchmark contains three internal and six external procedures, system and library routines are covered in the platform-specific configuration file. The SlaveStart() procedure is the entry point of the parallel algorithm and consists of a call to the FFT1D() procedure as well as some initialization and bookkeeping tasks. The FFT1D() procedure implements the six-step FFT algorithm described in Section 4.2.1, while the Transpose() procedure implements a blocked matrix transpose. The

first two procedures have to be internal since they contain calls to other internal procedures as well as barrier synchronizations, while the last procedure does not need to be internal. However, all inter-processor communication occurs in this procedure, hence the Transpose() procedure has to be internal in the parallel version of the fft benchmark used on the Cray T3E. In order to facilitate comparisons between the benchmarks on both platforms, this procedure is chosen to be internal as well. The super block optimization is quite effective, as the three internal procedures consist of 23, 42 and 43 basic blocks and 6, 9, and 1 super blocks, respectively.

The results of the experiments using the fft benchmark are summarized in Figures 5.8, 5.9, and 5.10. Figure 5.8 illustrates the results using a problem size of 64 K complex data points, while Figures 5.9 and 5.10 illustrate the results using problem sizes of 256 K and 1024 K, respectively. Note that a vertical baseline is provided for the base results in order to make identification of these results easier. The following paragraphs discuss the results illustrated in Figures 5.8, 5.9, and 5.10, respectively.

Using a problem size of 64 K complex data points, the corresponding array occupies 1 MB of memory, since double-precision floating-point values are 8 bytes long. Note that the fft benchmark aligns each row of the two-dimensional matrix on cache-line and pagesize boundaries, hence the matrix is usually slightly larger than 1 MB. There are three different arrays of this size: Two arrays are used as source and target during matrix transpose and contain the actual data matrix, the third array holds the roots-of-unity matrix. In addition, the first row of the roots-of-unity matrix is replicated by each thread.

All code and data segments of the fft benchmark should fit in the 4 MB second-level cache used in the Compaq XP1000 workstation, at least for the given problem size. According to [WOT+95], the first-level working set of the fft benchmark is one row of one of the matrices, i.e. the square of the problem size. The second-level working set is one partition of the whole data set, i.e. the size of the whole data set divided by the number of threads. In the case of multithreading, the individual threads share the cache resources, hence the working set as seen from the processor is probably identical to the whole data set.

An analysis of the data presented in Figure 5.8 yields the following results: The measured runtimes increase with the number of threads. On the one hand, each thread uses additional memory for the thread descriptor, thereby potentially increasing the number of cache misses. In the case of emulated multithreading, each thread descriptor requires 544 bytes of storage, at least on the current platform. On the other hand, the number of iterations for the main loop in the thread execution routine increases with the number of threads, thereby increasing the overhead associated with emulated multithreading.

Fig. 5.8. Results for the FFT Benchmark (64 K Complex Data Points)

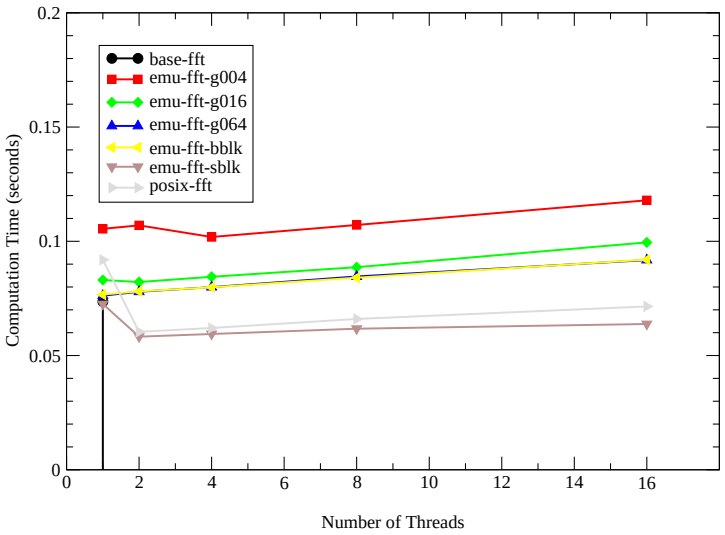
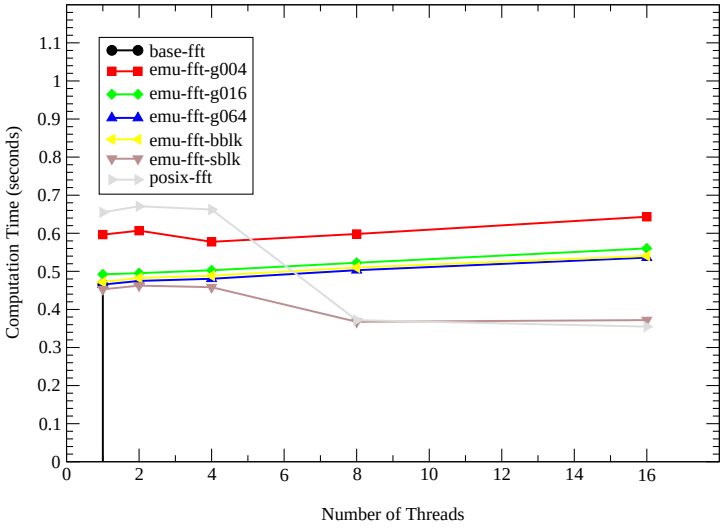
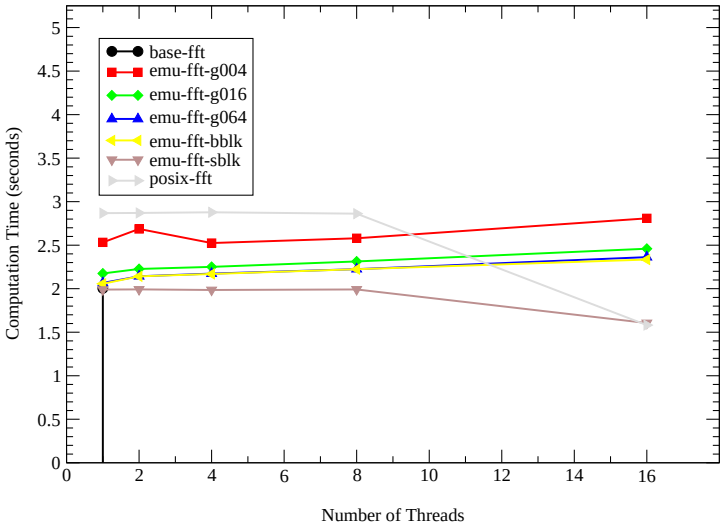


Fig. 5.9. Results for the FFT Benchmark (256 K Complex Data Points)



The only exception to this rule are the sblk and posix versions, where going from one to two threads decreases the runtime, afterwards the runtime increases with the number of threads. This behavior can be explained in the following way: If one thread is used, the transpose routine will transpose the whole matrix as a single block, accessing the source matrix in column order. As the individual rows are aligned on cache-line boundaries, 256 cache-lines are required to ensure that elements of the next column are already in

Fig. 5.10. Results for the FFT Benchmark (1024 K Complex Data Points)



the cache. Recall that the 21264 microprocessor uses a 64 KB two-way set associative first-level data cache with 64 byte cache-lines, i.e. 512 cache-lines in each set. As the cache uses a write-allocation protocol some cache-lines will be allocated by writes to the target matrix, although the writes occur row order, i.e. along cache-lines. Therefore transposing the matrix with a single thread is likely to cause cache misses in the first-level data cache. However, using p threads will eliminate these cache misses as each thread transposes the matrix in $\sqrt{n}/p \times \sqrt{n}/p$ blocks, thereby allocating fewer cache-lines.

Note that the above reasoning does not apply if several threads execute the Transpose() procedure simultaneously. If all threads transpose the matrix at the same time, the number of allocated cache-lines is comparable to the single-threaded case. This is supported by the fact that only the sblk version benefits from multiple threads: As the Transpose() procedure consists of a single large super block, no context switches occur during execution of the procedure, hence only one thread performs the transpose at any given time. All other versions using emulated multithreading perform context switches during matrix transpose. As the transpose is always preceded by a barrier synchronization, it is likely that all threads execute the Transpose() procedure simultaneously. However, if the procedure is external, all versions using emulated multithreading show the same behavior as the sblk version.

The runtimes of the g004, g016, g064, and bblk versions reflect the total number of instructions in the corresponding versions: The bblk and g064 versions have almost identical runtimes, the g016 version is slightly slower and the g004 version is significantly slower than the base version. Another interesting fact is that the sblk version using one thread has a runtime comparable to the base version, hence the overhead associated with emulated

multithreading is negligible in this case. Compared to the posix version, the sblk version is always faster, although the Tru64 operating system includes an efficient implementation of POSIX threads.

Using a problem size of 256 K complex data points, the three data matrices occupy slightly more than 4 MB of memory, hence the whole dataset will no longer fit in the second-level data cache. The size of the first- and second-level working sets quadruples as well, i.e. the first-level working set occupies 16 KB. The corresponding runtimes are illustrated in Figure 5.9. Apart from the overall increased runtime, the results are comparable to the ones using the smaller problem size. The exception are the sblk and posix versions as the reduction in runtime now occurs by using more than four as opposed to more than one thread. This behavior can be explained by the reduced number of misses in the first-level data cache, as the matrix is transposed in 64 K or less blocks if at least eight threads are used. Recall that each thread transposes one block locally and $p-1$ blocks to other threads, i.e. each thread transposes blocks of size $\sqrt{n}/p \times \sqrt{n}/p$, where n, p is the problem size and the number of threads, respectively.

Using a problem size of 1024 K complex data points, each of the three matrices occupies slightly more than 16 MB of memory. The size of the first- and second-level working sets quadruples again, i.e. the size of the first-level working set is equal to the size of the first-level data cache for the given problem size. The corresponding runtimes are illustrated in Figure 5.10. Apart from the overall increased runtimes, the results are comparable to the earlier ones using smaller problem sizes. Note that at least 16 threads are required for the runtime reduction mentioned above as the transpose uses 64 KB blocks in this case.

In summary, the results using the fft benchmark are quite encouraging: although there are no references to remote memory to be tolerated, the overhead associated with emulated multithreading is smaller than expected, especially if the sblk version is used. For all problem sizes and numbers of threads, the runtimes for the sblk version are smaller than the runtimes for the base version and are either smaller or comparable to the runtimes of the posix version.

5.5 LU

The project-specific configuration file for the lu benchmark contains three internal and six external procedures, system and library calls are covered in the platform-specific configuration file. The SlaveStart() procedure is the entry point of the parallel algorithm and contains only a call to the OneSolve() procedure. The latter procedure contains a call to the lu() procedure as well as some initialization and bookkeeping tasks. The lu() procedure implements the decomposition algorithm described in Section 4.2.2 by calling several

other procedures that perform the actual work on the individual blocks of the matrix.

As the entry point to the parallel algorithm, the SlaveStart() procedure has to be internal, it contains a call to an internal procedure as well. The OneSolve() procedure has to be internal as it contains a call to an internal procedure as well as some barrier synchronizations. Last but not least, the lu() procedure has to be internal as it contains some barrier synchronizations. For these three internal procedures, the super block optimization is quite effective as the procedures consist of 3, 19, and 48 basic blocks and 2, 5, and 6 super blocks, respectively.

The results of the experiments using the lu benchmark are summarized in Figures 5.11, 5.12, and 5.13. Figure 5.11 illustrates the results using a 512×512 matrix size, while Figures 5.12 and 5.13 illustrate the results using 1024×1024 and 2048×2048 matrix sizes, respectively. Note that a vertical baseline is provided for the base results in order to make identification of these results easier. The following paragraphs discuss the results illustrated in Figures 5.11, 5.12, and 5.13, respectively.

Using a 512×512 matrix size, the corresponding array occupies 2 MB of memory. The matrix is partitioned into blocks of size 16×16, each block occupies 2 KB of memory. According to [WOT+95], the first-level working set for the lu benchmark is one such block, while the second-level working set is one partition of the whole data set. In the case of multithreading, the individual threads share the cache resources, hence the working set as seen from the processor is probably identical to the whole data set. As the Compaq XP1000 workstation is configured with a 4 MB second-level cache, the whole data set will fit in this cache, at least for the given matrix size. In addition, a single block of the matrix will easily fit in the 64 KB first-level data cache provided by the 21264 microprocessor.

An analysis of the data illustrated in Figure 5.11 yields the following results: The runtimes increase with the number of threads for the multithreaded versions of the lu benchmark. This behavior was already observed during the analysis of the results for the fft benchmark and can be explained by the increased number of iterations for the main loop of the thread execution routine as well as the memory requirements for the additional thread descriptors. Note that all procedures that access the blocks of the matrix are external, i.e. no context switches occur during the execution of these procedures.

From all versions using emulated multithreading, the sblk version is the fastest, while the g004 version is the slowest. The g016, g064, and bblk versions show almost identical runtimes, indicating that most basic blocks contain no more than 16 instructions. In addition, the lu() procedure contains several external calls, hence the instruction blocks containing these calls are probably larger than 16 instructions as the restriction does not hold for these basic blocks.

Fig. 5.11. Results for the LU Benchmark (512 × 512 Matrix)

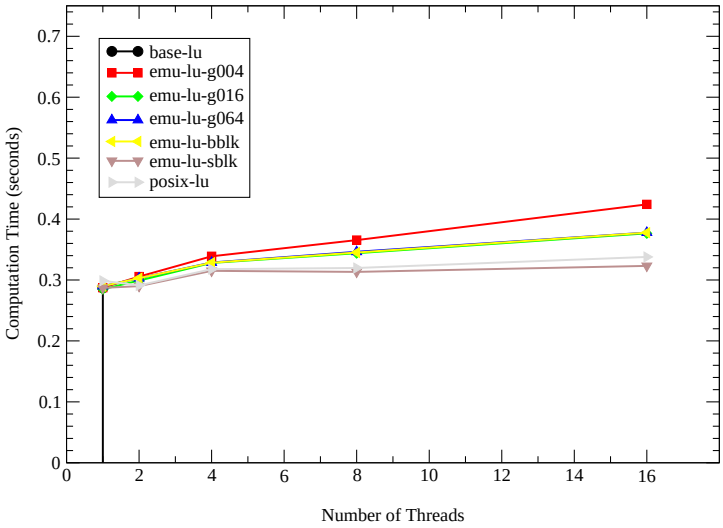
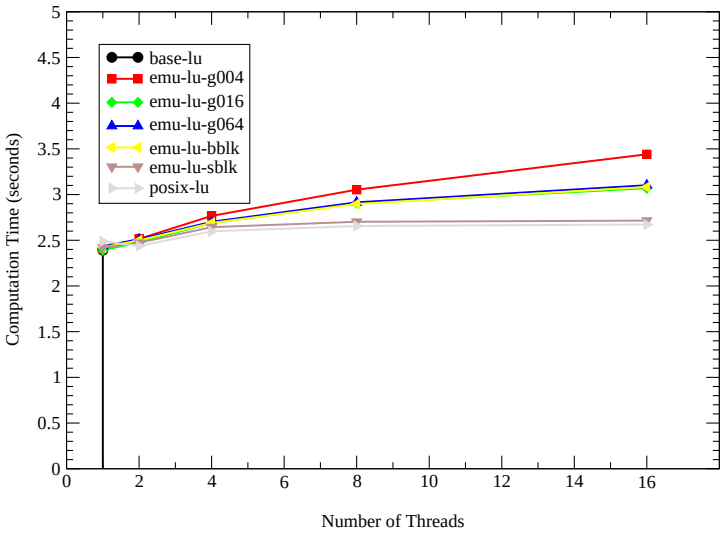
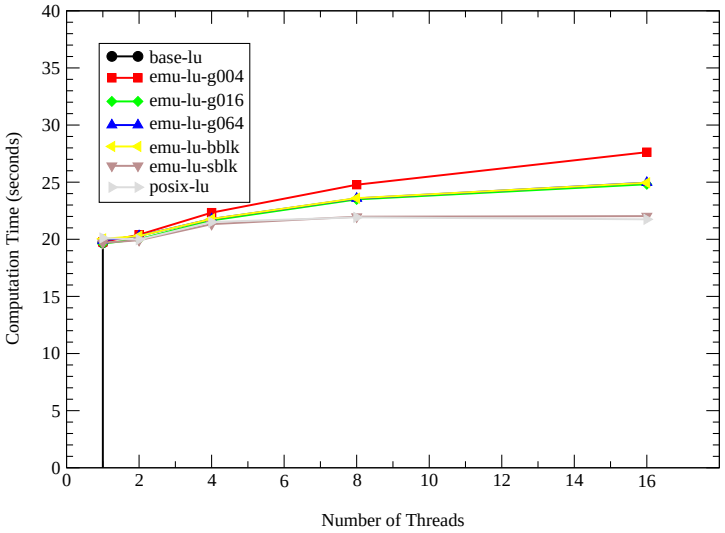


Fig. 5.12. Results for the LU Benchmark (1024 × 1024 Matrix)



The runtimes of the sblk version are slightly faster than the runtimes of the posix version and are almost identical to the base version in the single-threaded case. Although the runtimes increase with growing number of threads, the increase for the sblk version is slower than the increase for the other versions due to the smaller number of iterations for the main loop of the thread execution routine. Even with 16 threads, the sblk version is only 15 % slower than the base version.

Fig. 5.13. Results for the LU Benchmark (2048 × 2048 Matrix)



Using a 1024 × 1024 matrix, the corresponding array occupies 8 MB of memory, hence will no longer fit in the second-level cache. However, the size of a single block is unchanged, i.e. these 2 KB blocks will still fit in the first-level data cache. Apart from the overall increased runtimes, an analysis of the runtimes presented in Figure 5.12 yields the same results as for the smaller matrix size.

Using a 2048 × 2048 matrix, the corresponding array occupies 32 MB of memory, although the size of a single block is unchanged, i.e. these 2 KB blocks will still fit in the first-level data cache. Apart from the overall increased runtimes, an analysis of the runtimes presented in Figure 5.13 yields the same results as for the smaller matrix sizes.

In summary, the results of the experiments using the lu benchmarks are quite encouraging, although the versions using emulated multithreading are slightly slower than the base version: Even as there are no remote memory references to tolerate, the overhead associated with emulated multithreading is smaller than expected, especially if super block optimization is used. For all matrix sizes, the sblk version is slightly faster than the posix version, although the Tru64 operating system contains an efficient implementation of POSIX threads. In addition, the runtime of the sblk version in the single-threaded case is almost identical to the runtime of the base version.

5.6 Radix

The project-specific configuration file for the radix benchmark contains one internal and three external procedures, system and library calls are covered

in the platform-specific configuration file. The slavesort() procedure is the entry point of the parallel algorithm and implements the radix-sort algorithm as described in Section 4.2.3, the three external procedures are only used during initialization of the sort array. Apart from being the entry point, the slavesort() procedure has to be internal as it contains several synchronization points, e.g. barriers and spin waits. The super block optimization is quite effective for this procedure as the number of 160 basic blocks is reduced to only 26 super blocks.

The results of the experiments using the radix benchmark are summarized in Figures 5.14, 5.15, and 5.16. Figure 5.14 illustrates the results using a problem size of 256 K integers, while Figures 5.15 and 5.16 illustrate the results using problem sizes of 512 K and 1024 K integers, respectively. Note that a vertical baseline is provided for the base results in order to make identification of these results easier. The following paragraphs discuss the results illustrated in Figures 5.14, 5.15, and 5.16, respectively.

Using a problem size of 256 K integers, the corresponding array occupies 1 MB of memory, since the size of an int is 4 bytes. As the radix-sort algorithm does not sort in place, two of these arrays are needed, thereby occupying 2 MB of memory. In addition, each thread maintains a histogram of the local keys, the size of the corresponding array depends on the selected radix: Using a radix of 1024, each array occupies 4 KB of memory. Since the Compaq XP1000 workstation uses a 4 MB second-level cache, all code and data for the radix benchmark should fit in this cache, at least for the given problem size. According to [WOT+95], the first-level working set is one of the histograms, while the second-level working set is the local sort array of size n/p, where n, p are the total number of keys and threads, respectively.

An analysis of the data presented in Figure 5.14 yields the following results: The runtimes increase with the number of threads, although this effect is more pronounced than for the fft and lu benchmarks, at least for the g004, g016, g064, and bblk versions of the radix benchmark. For the sblk and posix versions, the increase in runtime is barely noticeable. Like before, this behavior can be explained by the increased number of iterations for the main loop in the thread execution routine for smaller instruction blocks, as well as the memory requirements for the additional thread descriptors. The effect is more pronounced for the radix benchmark in the absence of the super block optimization, as the slavesort() procedure contains several loops with a large number of iterations. For example, each thread sorts all the local keys once in each iteration, i.e. the corresponding loop is traversed n/p times in each intermediate sort. Other loops are used to copy or calculate the histogram, i.e. those loops are traversed r times in each iteration, where r is the radix size. Without super block optimization, at least one traversal of the main loop in the thread execution routine is required for each loop iteration. However, most of these loops do not contain any synchronization points, hence these

Fig. 5.14. Results for the Radix Benchmark (256 K Integers)

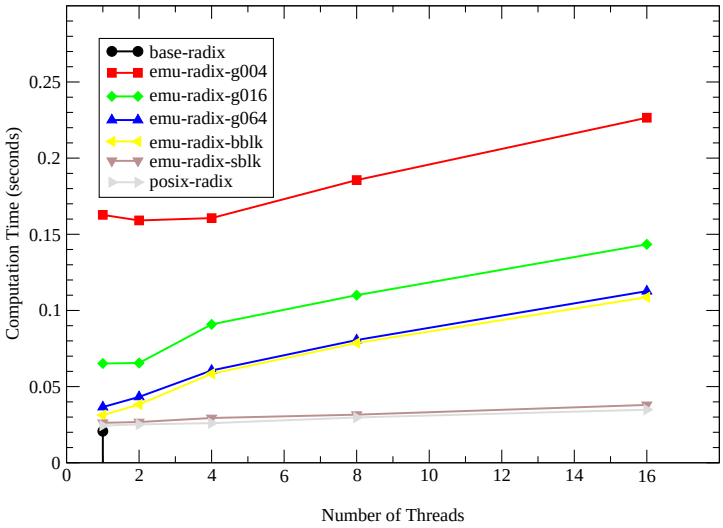
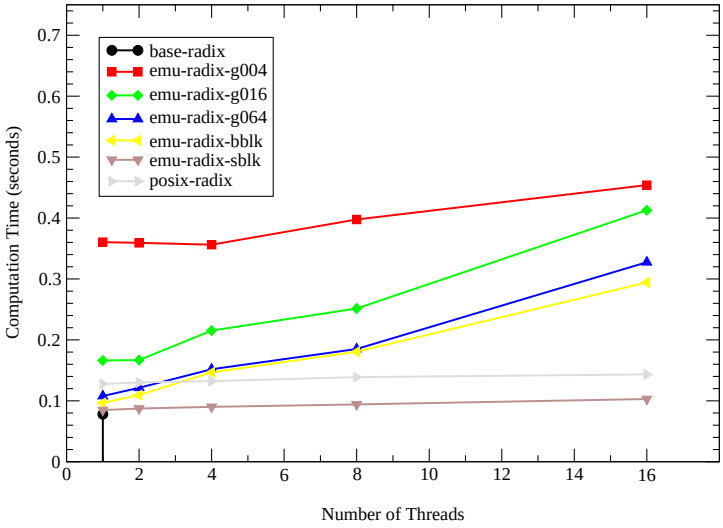


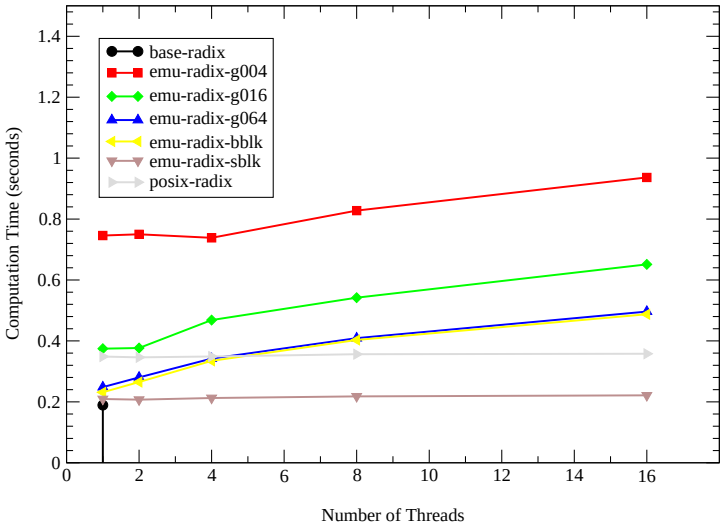
Fig. 5.15. Results for the Radix Benchmark (512 K Integers)



lops can be integrated into a single super block, thereby reducing the number of iterations of the main loop in the thread execution routine significantly.

The ratio between the runtimes of the g004, g016, g064, bblk, and sblk versions shows a well-known pattern: The sblk version is significantly faster than all other versions, the g064 and bblk versions have almost identical runtimes. The g016 version is slightly slower than the bblk version, while the g004 version is noticeably slower than the bblk version. This indicates that

Fig. 5.16. Results for the Radix Benchmark (1024 K Integers)



most of the basic blocks contain no more than 64 instructions, a fact that is supported by the statistics for the original assembler sources.

Using a problem size of 512 K integers, the two sort arrays occupy 4 M of memory hence the code and data of the radix benchmark will no longer fit in the second-level cache for the given problem size. However, the size of the histograms, i.e. 2 KB, is unchanged since the same radix is used for all problem sizes. Apart from the overall increased runtimes, an analysis of the runtimes illustrated in Figure 5.15 yields the same results as for the smaller problem size, although the posix version is now significantly slower than the sblk version.

Using a problem size of 1024 K integers, the two sort arrays occupy 8 M of memory, while the size of the histograms, i.e. 2 KB, is unchanged since the same radix is used for all problem sizes. Apart from the overall increased runtimes, an analysis of the runtimes illustrated in Figure 5.16 yields the same results as for the smaller problem sizes, the posix version is again significantly slower than the sblk version.

In summary, the results of the radix benchmark underline the importance of the super block optimization and the impact of the number of loop iterations on performance. Hence the super block optimization should be improved further, e.g. by using profiling information to guide the super block creation. Once again, the results are quite encouraging, as the sblk version is only slightly slower than the base version and is significantly faster than the posix version for the two larger problem sizes.

5.7 Ocean

The project-specific configuration file for the ocean benchmark contains three internal and 11 external procedures, system and library routines are covered in the platform-specific configuration file. The slave() procedure is the entry point for the parallel algorithm and performs one-time initialization as well as top-level flow control for the simulation. The slave2() procedure implements a single timestep of the ocean simulation that is divided into ten different phases, while the multig() procedure implements the multigrid solver used once during initialization and twice in each timestep.

These procedures have to be internal for several reasons: Apart from being the entry point, the slave() procedure contains five barrier synchronizations, one lock as well as one call to another internal procedure. The slave2() procedure contains ten barrier synchronizations, one lock as well as two calls to an internal procedure. The multig() procedure has to be internal as it contains five barrier synchronizations and one lock. Note that all those subroutines of the multigrid solver, that handle the actual data, are external. For these internal procedures, the super block optimization is very effective, as these procedures consist of 525, 785, and 28 basic blocks and 17, 16, and 12 super blocks, respectively.

The results of the experiments using the ocean benchmark are summarized in Figures 5.17, 5.18, and 5.19. Figure 5.17 illustrates the results using an ocean of 130×130 grid points, while Figures 5.18 and 5.19 illustrate the results using oceans of 258×258 and 514×514 grid points, respectively. Note that a vertical baseline is provided for the base results in order to make identification of these results easier. The following paragraphs discuss the results illustrated in Figures 5.17, 5.18, and 5.19, respectively.

Using an ocean of 130×130 grid points, a corresponding array occupies approximately 132 KB of memory. The ocean benchmark uses a fairly large number of these arrays, i.e. 25. Apart from these and several smaller arrays the two arrays used as input for the multigrid solver consist of seven separate arrays, one for each level. Note that the number of levels is equal to the binary logarithm of the problem size, i.e. 128. Even for the smallest problem size, the dataset of the ocean benchmark will therefore not fit into any of the caches in the Compaq XP1000 workstation. According to [WOT+95], the first-level working set of the ocean benchmark consists of a few subrows, i.e. a few KB, while the second-level working set is one partition of the whole data set, i.e. the size of the whole data set divided by the number of threads. In the case of multithreading, the individual threads share the cache resources, hence the working set as seen from the processor is probably identical to the whole data set. Although the second-level working set will not fit in any of the caches, the first-level working set will fit in the first-level data cache.

An analysis of the runtimes presented in Figure 5.17 yields the following results: The runtimes increase with the number of threads, although using four threads is slower than using eight threads for all versions using emulated

Fig. 5.17. Results for the Ocean Benchmark (130 × 130 Ocean)

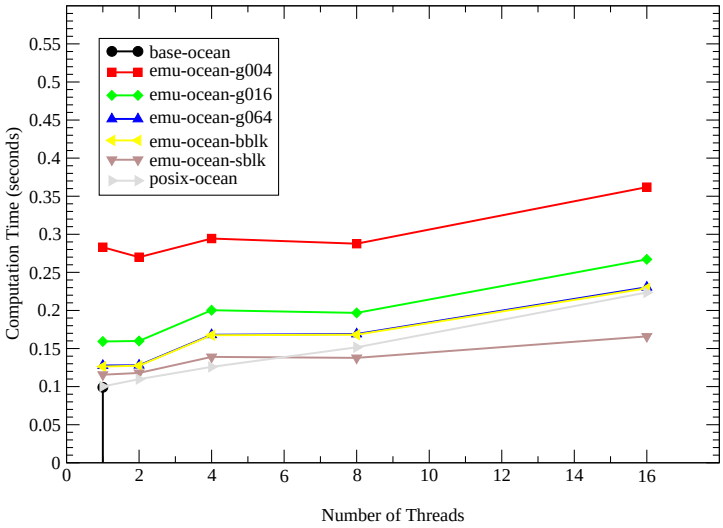
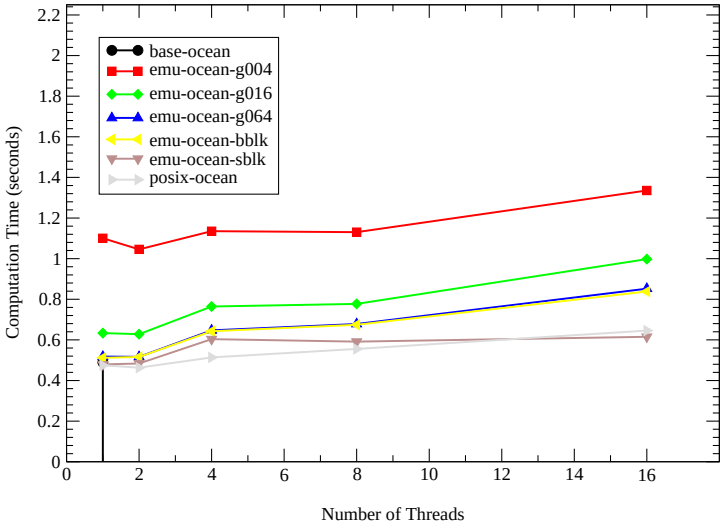
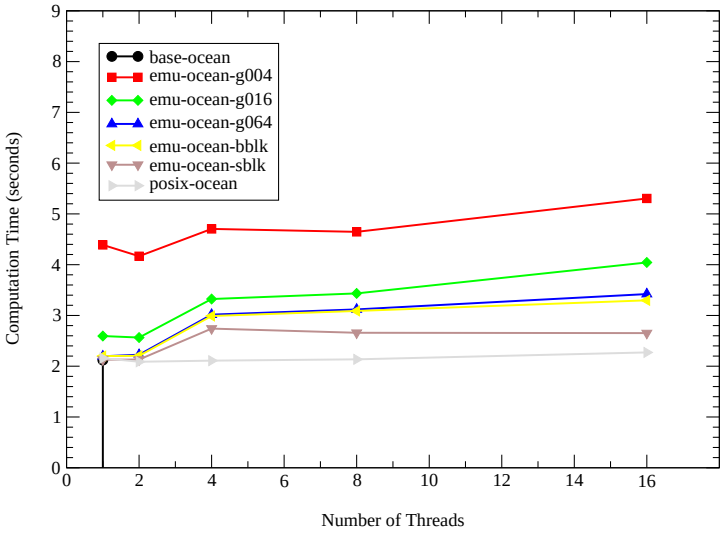


Fig. 5.18. Results for the Ocean Benchmark (258 × 258 Ocean)



multithreading. This sudden increase in runtime is caused in the multigrid solver, as the time spent in the solver increases significantly if four threads are used. Subtracting the times spent in the multigrid solver from the overall runtimes yields the slight increase in runtime observed for the other benchmarks. However, the source of the sudden increase in the amount of time spent in the multigrid solver is unclear: As the posix version is not affected, this behavior seems to be caused by emulated multithreading. As only the top-level routine

Fig. 5.19. Results for the Ocean Benchmark (514 × 514 Ocean)



of the multigrid solver is internal and all subroutines are external, it is unlikely that an increase in the number of cache misses is responsible. Another candidate would be the lock in the multigrid solver that is used to sum up the local errors. However, a detailed analysis of the cache misses will allow to rule out the former and further investigate this phenomenon. Unfortunately, the performance counters in the 21264 microprocessor do not support the corresponding events, hence it is not possible to perform this analysis on the current platform. .

Apart from the phenomenon described above, the runtimes of the g004, g016, g064, bblk, and sblk versions follow a well-known pattern: For all numbers of threads, the sblk version is the fastest, while the bblk and g064 versions have almost identical runtimes. The g004 version is noticeably slower than the bblk version and the g016 version is slightly slower than the bblk version. The sblk version is slower than the posix version as long as less than eight threads are used, but is significantly faster for 8 and 16 threads.

Using an ocean of 258 × 258 grid points, each of the 25 arrays occupies approximately 520 KB of memory, i.e. about 13 MB total. The two arrays used as input for the multigrid solver have eight levels and occupy approximately 7 MB together. While the second-level working set will not fit in any of the caches, the first-level working set occupies several KB and will fit in the first-level data cache. Apart from the overall increased runtime, an analysis of the runtimes presented in Figure 5.18 yields the same results as for the smaller problem size, although the posix version is always faster than the sblk version.

Using an ocean of 514 × 514 grid points, each of the 25 arrays occupies approximately 2 MB, i.e. 50 MB total. The two arrays used as input for the multigrid solver have nine levels and occupy approximately 32 MB together.

Again, the second-level working set will not fit in any of the caches, while the first-level working set may still fit in the first-level data cache as each row occupies about 4 KB. Apart from the overall increased runtime, an analysis of the runtimes presented in Figure 5.19 yields the same results as for the smaller problem sizes, although the advantage of the posix version increases slightly.

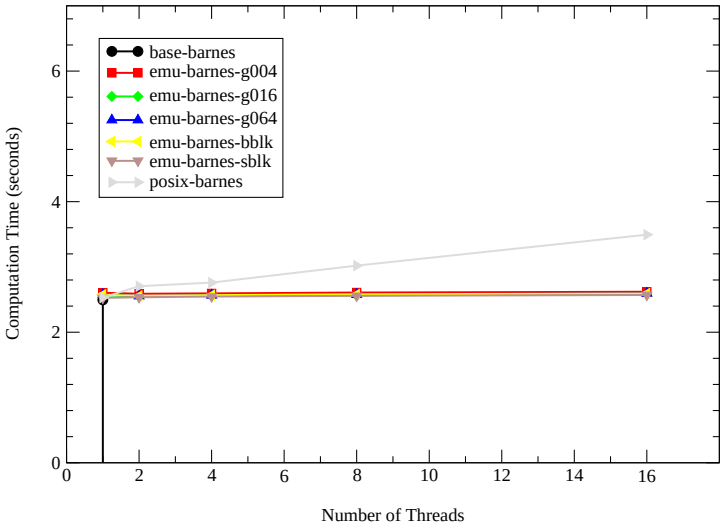
In summary, the results of the ocean benchmark are a bit disappointing, since the sblk version is slower than the posix version. The increase in runtime by using four threads has yet to be explained, a detailed analysis of the cache behavior will provide useful hints in this regard.

5.8 Barnes

The project-specific configuration file for the barnes benchmark contains five internal and five external procedures, system and library routines are covered in the platform-specific configuration file. The SlaveStart() procedure contains some assignments, a small loop that calls the internal StepSimulation() procedure for the selected number of timesteps as well as a call to the find_my_initial_bodies() procedure. The latter procedure performs the initial distribution of particles to threads and consists of a loop that updates the particle pointers in the corresponding local array of the thread. The number of loop iterations for all threads is equal to the number of particles. The StepSimulation() procedure implements a single timestep of the particle simulation and consists of a sequence of internal and external calls that implement the different phases of the algorithm described in Section 4.2.5. The final loop is used to update the local particles, hence the number of loop iterations across all threads is equal to the number of particles. The maketree() procedure constructs the particle tree and consists of a single loop that calls the loadtree() procedure for each of the local particles. The hackcofm() procedure determines the center of mass for all leafs and cells in the particle tree that belong to the current thread. The procedure consists of two loops that are used to traverse the corresponding local arrays such that the number of loop iterations across all threads is equal to the number of particles.

Apart from being the entry point, the SlaveStart() procedure has to be internal as it contains two calls to other internal procedures. Due to a barrier synchronization, the find_my_initial_bodies() procedure has to be internal as well. The StepSimulation() procedure has to be internal as it contains two barriers, one lock, as well as calls to other internal procedures. In the same way, the maketree() procedure has to be internal as it contains two barriers, one lock, and a call to another internal procedure. Last but not least, the hackcofm() procedure contains a spin wait and therefore has to be internal. For these internal procedures, the super block optimization is quite effective as the procedures consist of 6, 11, 65, 14, and 19 basic blocks and 5, 2, 9, 4, and 9 super blocks, respectively.

Fig. 5.20. Results for the Barnes Benchmark (16 K Particles)



The results of the experiments using the barnes benchmark are summarized in Figures 5.20, 5.21, and 5.22. Figure 5.20 illustrates the results using a problem size of 16 K particles, while Figures 5.21 and 5.22 illustrate the results using problem sizes of 64 K and 256 K particles, respectively. Note that a vertical baseline is provided for the base results in order to make identification of these results easier. The following paragraphs discuss the results illustrated in Figures 5.20, 5.21, and 5.22, respectively.

Using a problem size of 16 K particles, the barnes benchmark allocates approximately 2 MB of memory for the global particle array, another 2 MB and 4 MB for the local cell and leaf arrays, as well as approximately 256 KB for the local cell and leaf pointer arrays. According to [WOT+95], the first-level working set of the barnes benchmark is the tree data for one particle, i.e. the leaf that contains the particle as well as all cells on the path from the root cell to that leaf. As the height of the tree is logarithmic and the size of a leaf and cell is 148 and 168 bytes, the first-level working set is a few KB large and will therefore fit in the first-level data cache. The second-level working set is one partition of the whole data set, i.e. probably the whole data set in the case of multithreaded execution. As the Compaq XP1000 workstation is configured with a 4 MB second-level cache, the second-level working set will not fit in any of the caches.

An analysis of the runtimes presented in Figure 5.20 yields the following results: The runtimes increase with the number of threads, although the difference is almost negligible for the current problem size. The relation between the different versions of the barnes benchmark shows the same well-known pattern as for the other benchmarks: The sblk version is the fastest, the bblk and g064 versions have almost identical runtimes, while the g004 version is the

Fig. 5.21. Results for the Barnes Benchmark (64 K Particles)

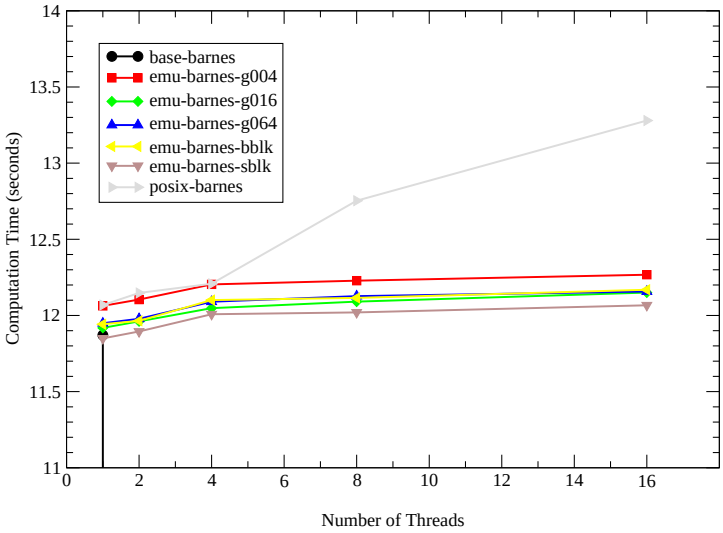
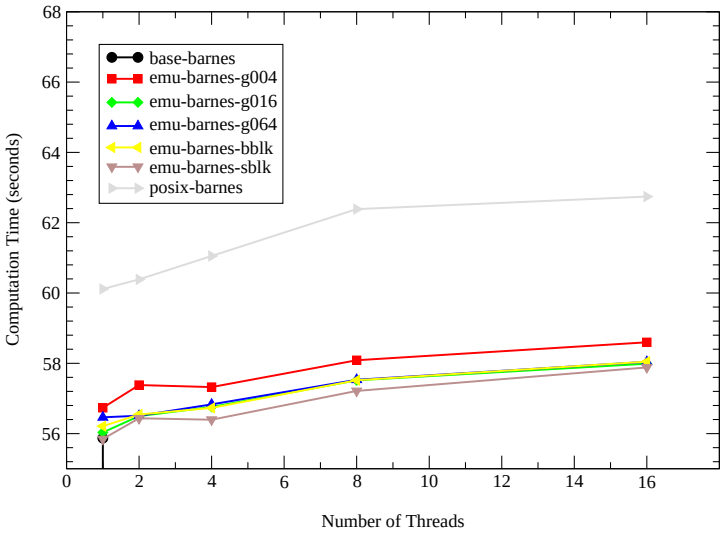


Fig. 5.22. Results for the Barnes Benchmark (256 K Particles)



slowest. However, the g016 version is different as it is slightly faster than the bblk version. The posix version is slower than all other versions if more than one thread is used. This performance degradation is caused by a significant increase of time spent in the force calculation phase. The sblk optimization is quite effective, as the runtimes for the loop that updates the individual particles is independent of the number of threads, while it increases significantly in the absence of super block optimization due to the larger number of

context switches. Compared to the overall runtime, the overhead associated with emulated multithreading is almost negligible.

Using a problem size of 64 K particles, the barnes benchmark allocates approximately 7 MB of memory for the global particle array, another 8 MB and 16 MB for the local cell and leaf arrays as well as approximately 1 MB for the local cell and leaf pointer arrays. The size of the first-level working set increases only slightly as the size of the working set only increases with the logarithmic height of the particle tree. While the first-level working set will still fit in the first-level data cache, the second-level working set will not fit in any of the caches. Apart from the overall increased runtimes, an analysis of the runtimes presented in Figure 5.21 yields the same results as for the smaller problem size, although the distance of the posix version increases.

Using a problem size of 256 K particles, the barnes benchmark allocates approximately 28 MB of memory for the global particle array, another 16 MB and 32 MB for the local cell and leaf arrays as well as approximately 4 MB for the local cell and leaf pointer arrays. The size of the first-level working set increases only slightly as the size of the working set only increases with the logarithmic height of the particle tree. While the first-level working set will still fit in the first-level data cache, the second-level working set will not fit in any of the caches. Apart from the overall increased runtimes, an analysis of the runtimes presented in Figure 5.22 yields the same results as for the smaller problem sizes, although the distance of the posix version increases even more.

In summary, the results for the barnes benchmark are quite encouraging, as all versions using emulated multithreading are faster than the posix version. Compared to the base version, the overhead introduced by emulated multithreading is negligible.

5.9 FMM

The fmm benchmark is by far the most complex benchmark used during the evaluation of emulated multithreading. The project-specific configuration file for the fmm benchmark contains 32 internal and 25 external procedures, system and library routines are covered in the platform-specific configuration file. All of these procedures have to be internal as the either contain a synchronization point, e.g. barriers or locks, or a call to another internal procedure. Due to the large number of synchronization points and internal calls, the super block optimization is not very effective for the internal procedures in the fmm benchmark: These procedures consist of 532 basic blocks and 253 super blocks, respectively. The rather large number of super blocks is probably the cause for the highest increase in the number of instructions encountered so far.

The results of the experiments using the fmm benchmark are summarized in Figures 5.23 and 5.24. Figure 5.23 illustrates the results using a problem

size of 16 K particles, while Figure 5.24 illustrates the results using a problem size of 64 K particles. The results using a problem size of 256 K particles have been omitted, as the allocated memory exceeds the installed physical memory in the Compaq XP1000 workstation, i.e. the operating system starts to swap. Note that a vertical baseline is provided for the base results in order to make identification of these results easier. The following paragraphs discuss the results illustrated in Figures 5.23 and 5.24, respectively.

Using a problem size of 16 K particles, the fmm benchmarks allocates approximately 1.5 MB for the global particle array, 0.5 MB for the local particle pointer arrays as well as 18 MB for the local cell arrays. According to [WOT+95], the first-level working set for the fmm benchmark are the expansion terms, which have a fixed size of less than 640 bytes. As the 21264 microprocessor used in the Compaq XP1000 workstation contains a 64 KB first-level data cache, the first-level working set will easily fit in this cache. The second-level working set is one partition of the whole data set, i.e. probably the whole data set in the case of multithreaded execution. Even for the smallest problem size, the size of the working set exceeds the size of the 4 MB second-level cache used in the Compaq XP1000 workstation.

An analysis of the runtimes presented in Figure 5.23 yields the following results: Compared to the overall runtime, the increased runtime for using multiple threads is almost negligible. The runtimes of the g004, g016, g064, bblk, and sblk versions follow a well-known pattern: The sblk version is significantly faster than all other versions, the g064 and bblk versions have almost identical runtimes. The g004 version is significantly slower than the bblk version, while the g016 version is slightly slower than the bblk version. However, the difference between the individual versions is larger than for all other benchmarks. In addition, the overhead associated with emulated multithreading is rather high. Even the sblk version is more than two times slower than the base version. Nevertheless, the posix version is significantly slower than the sblk version if more than one thread is used. Investigating the individual components of the runtime reveals that almost all time is spent in computing the interactions between particles and cells. Unfortunately, the large number of internal procedures makes the analysis rather difficult. A detailed analysis of the cache behavior will probably reveal the reason for the large overheads in the case of the fmm benchmark. As the increase in runtime due to a larger number of threads is almost negligible, it is unlikely that the large overhead is caused by context switches.

Using a problem size of 64 K particles, the fmm benchmarks allocates approximately 6 MB for the global particle array, 2 MB for the local particle pointer arrays as well as 72 MB for the local cell arrays. As the first-level working set is of fixed size and occupies less than 640 bytes, this working set will still fit in the first-level data cache. Like before, the second-level working set is too large to fit in any of the caches in the Compaq XP1000 workstation. Apart from the overall increased runtime, an analysis of the

Fig. 5.23. Results for the FMM Benchmark (16 K Particles)

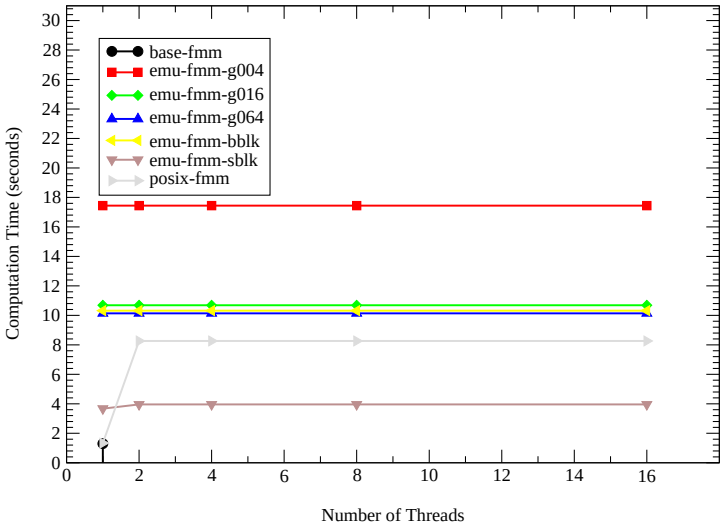
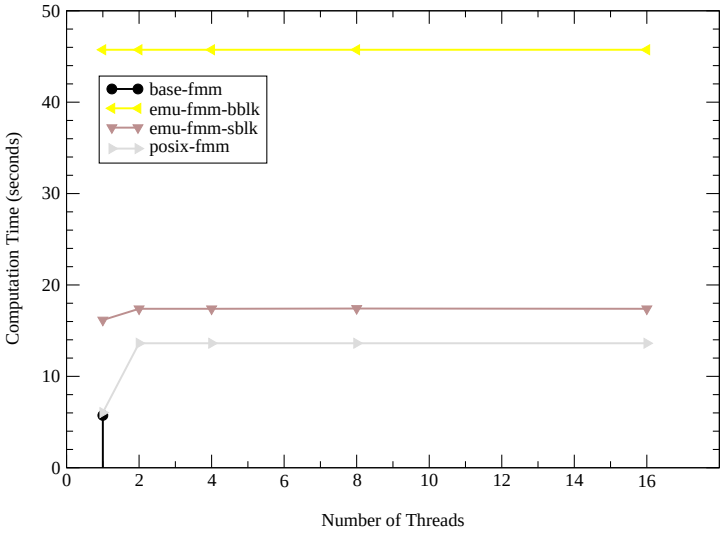


Fig. 5.24. Results for the FMM Benchmark (64 K Particles)



runtimes presented in Figure 5.24 yields the same results as for the smaller problem size, although the overhead for the bblk and sblk versions is even more pronounced and the sblk version is now slower than the posix version. Note that the results for the g004, g016, g064 versions were omitted due to instabilities of the corresponding executables.

In summary, the results for the fmm benchmark are even worse than the results for the barnes benchmark: The overhead for the versions using

emulated multithreading is very large, even the sblk version is significantly slower than the base version. For the larger problem size, the sblk version is slower than the posix version for all numbers of threads. The reason for the large overhead is unclear, as the size of the application code complicates the analysis of the results. A detailed analysis of the cache behavior would provide useful hints in this regard. Unfortunately it is not possible to perform this analysis on the Compaq XP1000 workstation due to the restricted set of events supported by the performance counters of the 21264 microprocessor.

5.10 Summary

The evaluation of emulated multithreading on the Compaq XP1000 workstation revealed that the overhead associated with emulated multithreading is smaller than expected, especially if the super block optimization is used. In fact, the sblk version is faster than the posix version in most cases. As the Tru64 operating system contains an efficient implementation of POSIX threads, this result demonstrates that emulated multithreading is feasible and that the efforts to reduce the overhead have paid off. In addition, the current implementation of emulated multithreading provides several opportunities for further improvements, e.g. by improving the code conversion process or the emulation library.

Recall that the current implementation of emulated multithreading is targeted at tolerating the latency of remote memory references in massively parallel processors. As these events do not occur on a single-processor workstation like the Compaq XP1000, there is nothing to be gained by latency tolerance on this platform. Therefore the results of the evaluation reflect the overhead associated with emulated multithreading and provide a good foundation for the evaluation on massively parallel processors.

However, the evaluation of emulated multithreading on the Compaq XP1000 workstation should be extended by performing a detailed analysis of the cache behavior. Such an analysis would provide useful insights, especially in connection with an extension of emulated multithreading to cover the latency of main memory accesses as well. For example, the sblk version of the fft benchmark benefits from the reduced number of cache misses due to the smaller working sets if multiple threads are used. As the latency associated with references to main memory is on the order of hundreds of cycles, emulated multithreading could be extended to tolerate these results if the context switch locations are chosen carefully.

6. Evaluation : Cray T3E

As described in Chapter 2, emulated multithreading is designed to tolerate long latency events by using fine-grained multithreading and asynchronous communication. The current implementation described in Chapter 3 is targeted at tolerating remote memory accesses in massively parallel processors, since such accesses incur a large latency and are easily identifiable. Emulated multithreading was evaluated on a single-processor workstation in order to determine the characteristics of emulated multithreading as well as the associated overhead. The corresponding results are presented in Chapter 5. This chapter evaluates emulated multithreading on a massively parallel processor, i.e. the Cray T3E. This evaluation was supported by grants for computing time from the John-Neumann Institute of Computing (NIC) at the Forschungszentrum Jülich and the Höchstleistungsrechenzentrum (HLRS) Stuttgart.

The two computing centers mentioned above provided access to three different models of the Cray T3E: A 512-processor T3E-1200 and a 512-processor T3E-600 are installed at the NIC, while a 512-processor T3E-900 is installed at the HLRS. All experiments performed during the evaluation were executed on the Cray T3E-1200 at the NIC, while the T3E-900 at the HLRS was primarily used during development and test of the current implementation. The T3E-600 installed at the NIC is reserved for batch jobs with a large number of processors and was therefore not used in the evaluation.

Emulated multithreading is evaluated on the Cray T3E massively parallel processor using five of the six benchmarks described in Chapter 4, i.e. fft, lu, radix, ocean and barnes. The fmm benchmark is not used in the evaluation due to several race conditions in the ported application that lead to frequent instabilities, especially for larger numbers of processors. A detailed description of these race conditions as well as two workarounds are provided in Section 4.2.6. For each benchmark, several experiments are performed in order to assess the performance of emulated multithreading on a wide range of problem sizes, processor numbers, and thread numbers. These results are compared with the results obtained from single-threaded execution on the same range of problem sizes and processor numbers.

Section 6.1 describes the architecture of the Cray T3E and emphasizes the E-register mechanism that is used for remote memory accesses, while Section

6.2 covers the experimental methodology used in the evaluation. Sections 6.3, 6.4, and 6.5 discuss the evaluation of emulated multithreading using the fft, lu, and radix benchmarks, respectively. Sections 6.6 and 6.7 cover the evaluation of multithreading using the ocean and barnes benchmarks. In contrast to the previous chapter, the impact of the code conversion process on the assembler sources of the benchmarks is omitted since the facts and reasoning from the Compaq XP1000 platform also apply to the Cray T3E platform.

6.1 Cray T3E

The Cray T3E [Oed96] is based on the earlier Cray T3D [KFW94] and represents the second-generation of massively parallel processors from Cray Research. The Cray T3E supports up to 2048 processing elements based on the Alpha 21164 processor described in Section 6.1.1. The global addressable memory is physically distributed among the processing elements and is covered in Section 6.1.2. The individual processing elements are connected by a bidirectional three-dimensional torus network that is described in Section 6.1.3. The I/O architecture of the Cray T3E is based on SCI (Scalable Coherent Interconnect) and is described in Section 6.1.4. Finally, Section 6.1.5 describes the unicos/mk operating system as well as the programming environments that are available on the Cray T3E. The information was this section are gathered from several sources, i.e. [Oed96][AXP96c][ST96][Sco96]

The Cray T3E is available in four different models. The major difference between these models is the clock frequency of the microprocessors used in each processing element: The -600, -900, -1200, and -1350 models use clock frequencies of 300, 450, 600, and 675 MHz, respectively. The system clock frequency is the same for all models, i.e. 75 MHz. Apart from the different models, the Cray T3E is available in two different versions, i.e. liquid cooled and air cooled.

The liquid cooled version supports up to eight chassis, where each chassis contains up to 256 user processing elements, 16 support processing elements, as well as one clock module for a total of 2048 user and 128 support processing elements. Note that the number of processing elements can be increased in steps of eight, i.e. it is not necessary to double the number of processing elements in each step. In each case, two chassis share a heat exchange unit that connects the primary fluorinert and the secondary water cooling circuits.

The air cooled version supports up to six chassis, where each chassis contains up to 20 or 24 user or support processing elements and up to one clock module for a total of 128 user and 8 support processing elements. Note that the number of processing elements can be increased in steps of four instead of eight.

6.1.1 Processor

Each processing element in the Cray T3E is based on an Alpha 21164 processor running at a clock frequency of 300, 450, 600, or 675 MHz, depending on the Cray T3E model. Note that the processor clock frequency is always a multiple of the 75 MHz clock frequency used by the system logic and network. The 21164 microprocessor is an implementation of the Alpha architecture, a 64 bit architecture designed by Digital Equipment Corporation. A detailed description of this architecture as well as all implementations can be found in Appendix A. The Alpha architecture uses a split register file, there are 32 integer and 32 floating-point registers, each 64 bit wide.

The 21164 microprocessor is a super-scalar processor that can issue up to four instructions (two integer, two floating-point) in each cycle. The two integer execution pipelines support all arithmetic instructions on 32 bit and 64 bit integer operands, integer control instructions as well as load and store instructions for integer and floating-point operands. The two floating-point execution pipelines support all arithmetic instructions on single- and double-precision floating-point operands as well as floating-point control instructions. Note that floating-point division is handled by a non-pipelined divider that is associated with one of the floating-point execution pipelines.

The 21164 microprocessor contains first-level data and instruction caches as well as an unified second-level cache. The first-level data cache is an 8 KB direct mapped cache with 32 byte cache-lines and uses a write-through, read-allocate protocol. The data cache is dual-ported to allow two independent accesses in each cycle. The first-level instruction cache is an 8 KB direct-mapped cache with 32 byte cache-lines. The second-level cache is a 96 KB 3-way set-associative cache using 32 or 64 byte cache-lines and a write-back, write-allocate protocol. Each instruction that hits in the data cache incurs a latency of two cycles, while instructions that hit in the unified cache incur a latency of eight cycles or more.

The internal caches are backed by an optional external cache. This third-level cache is a direct-mapped cache ranging from 1 to 64 MB in size and using 32 or 64 byte cache-lines. However, the processing elements in the Cray T3E do not use this option in order to decrease the latency of main memory accesses. Stream buffers are used instead to increase the performance of non-unit-stride accesses. The memory system of the processing elements is described in Section 6.1.2. In order to improve the performance of load and store instructions, the 21164 contains a miss address file that buffers and merges outstanding loads as well as a write buffer that buffers and merges stores. Detailed information about the internal architecture of the 21164 can be found in Section A.3.6.

The Alpha architecture defines a 64 bit virtual address space, implementations are required to support at least 43 bit virtual address space and check the remaining bits to be zero. The 21164 microprocessor implements the minimum 43 bit address space as well as a 40 bit physical address space. In

contrast to the 21064 microprocessor used in the Cray T3D [ACK+95], the size of the physical address space is large enough to cover the total amount of memory supported by the Cray T3E, i.e. 4 TB. The physical address space is divided into two parts by the most significant address bit: The lower part is cached by the internal caches and contains the local memory of a processing element, the upper part is uncached and contains the memory-mapped registers used to access remote memory.

6.1.2 Memory

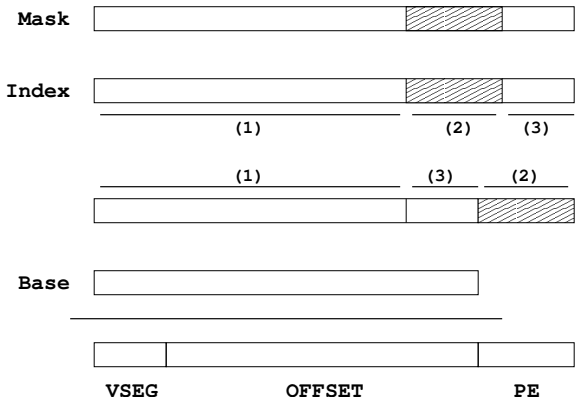
The Cray T3E provides a global address space with local consistency although the memory is physically distributed among the processing elements. Each processing element contains a local memory that resides in the cache address space of the corresponding processor. The local memory system consists of eight independent single-word (64 bit) banks of DRAM, each pair of banks is controlled by a bank control chip. These are connected to the system control logic via a 32 bit data bus, i.e. the bandwidth of the four memory busses is equal to the bandwidth of the 128 bit processor bus. Note that the number of memory banks allows a quasi-parallel access to 64 byte cache-lines.

Instead of an external third-level cache, the processing elements in the Cray T3E use a stream buffer [Jou90] to prefetch cache-lines starting on cache miss addresses: After two misses to consecutive cache-lines, a stream buffer is allocated and starts to prefetch data or instructions by initiating accesses starting at the cache miss address. The stride between the individual accesses is calculated as the difference between the first two addresses that missed in the cache and activated the stream buffer [BC91a]. Note that the misses do not have to be contiguous, the stream detection logic maintains the history of the eight last cache misses in order to identify streams. In contrast to caches, stream buffers are useful for non-unit-stride accesses, e.g. accesses to vectors or matrices. The system control logic in the Cray T3E supports six independent stream buffers, where each stream buffer contains up to two consecutive cache-lines, e.g. 128 byte.

The stream buffers are completely managed in hardware, but software can provide hints to guide the allocation of stream buffers. Unfortunately, the initial version of the system control logic had a serious flaw in connection with the stream buffers and cache consistency [Cra96], hence most installations of a Cray T3E-600 disable the stream buffers by default. A detailed description of this flaw as well as possible workarounds are included in Section B.3. Cache coherency between the internal caches, stream buffers, and main memory is maintained in hardware, using the flush coherency protocol supported by the 21164 microprocessor. An external backmap, i.e. a copy of the second-level cache tags, is used to invalidate cache-lines or stream buffer entries in the advent of remote writes.

Although the memory is physically distributed among the processing elements, each processor can access all memory locations on any of the other

Fig. 6.1. Global Address Calculation - Part I



processing elements. Note that these remote accesses are performed without any involvement of the remote processor. Although the memory is globally addressable, there is no shared address space: Each memory location is identified by the address of the location in the address space of the corresponding processing element as well as the number of this processing element. Access to remote memory is performed by means of the so-called E-registers. All remote transfers occur between these E-registers and the main memory of the remote processing element. The E-registers reside in the uncached region of the address space and can be read and written by the processor via uncached load and stores, limiting the bandwidth between these registers and the processor to 600 MB/s. Note that cached load and store instructions can be performed at up to 1200 MB/s.

There are 640 E-registers, 512 of these are available to applications and 128 are reserved for system use. The large number of E-registers facilitates pipelining of remote memory accesses by increasing the number of outstanding operations. These E-registers are used in two different ways: They can either be used as a source-and-destination E-register (SADE) that provides the source or destination operands for global get and put operations, respectively. Besides, they can be used as part of a more-operands-block of E-registers (MOBE) that provides additional operands during global address generation. Given a 50 bit address index, the mask stored in the first E-register of the block is used to extract those bits, that form the number of the remote processing element. The indicated bits are compacted into a zero-extended 12 bit number, i.e. the number is large enough to support the maximum number of 2048 user and 128 support processing elements. The remaining bits are compacted and added to the base address stored in the second E-register of the block. The six most significant bits of the result represent a virtual segment, while the remaining 32 bits represent an offset

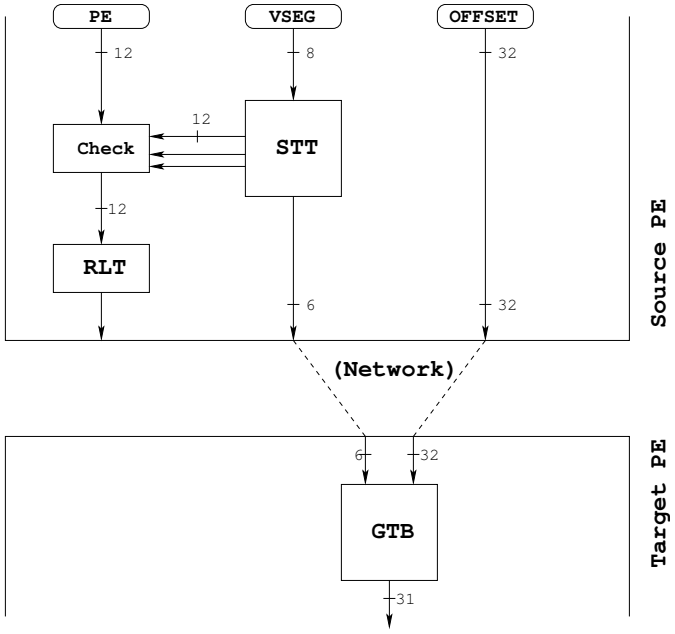
within that segment. These operations are performed by the hardware centrifuge and are illustrated in Figure 6.1. Note that the hardware centrifuge supports sophisticated data distribution models, e.g. distributing an array to other processing elements on cache-line boundaries.

The access to global memory is initiated by a store to a specific address in the uncached region of the address space: The address specifies the type of access as well as the number of the source-and-destination E-register to use during the actual data transfer. The written data specifies the number of the more-operands-block of E-registers to use during address translation, as well as an address index that is translated into a global virtual address via the hardware centrifuge. Note that the more-operand block of E-registers is initialized only once. All subsequent accesses to the corresponding distributed array can be performed directly.

The second part of the global address translation is depicted in Figure 6.2. After the virtual segment, segment offset, and virtual processing element have been determined by means of the hardware centrifuge, these values are translated into a global virtual address. Such an address consists of a global segment, segment offset, and routing informations. The virtual segment number is translated into a global segment number by the segment translation table (STT). For each virtual segment, this table stores the number of the corresponding global segment, the number of the base processing element in the current partition, as well as protection information. The number of the base processing element is added to the virtual processing element number determined by the hardware centrifuge, yielding the logical number of the target processing element. This number is checked for access violations using the protection information and the size of the current partition stored in the segment translation table. Afterwards the logical processing element number is translated into a physical processing element number, i.e. routing information, by means of the routing lookup table (RLT).

The physical processing element numbers denote the position of the corresponding processing element in the three-dimensional torus network, while the logical number is used to identify all processors that are visible to the operating system and the virtual number represents the processing elements inside the current partition. The virtual numbers range from 0 to $p-1$, where p is the number of processing elements in the current partition, while the logical numbers range from 0 to $n-1$, where n is the number of processors visible to the operating system. The physical number determines the x, y, and z coordinates of the processing element in the three-dimensional torus network. The number of physical processing elements may be larger than the number of logical processing elements due to redundant processing elements used to replace failed elements. The segment translation table at the source processing element ensures that only authorized global segments on authorized processing elements are accessed, while the routing lookup table allows

Fig. 6.2. Global Address Calculation - Part II



redundant processing elements to be mapped in place of failed processing elements.

After generating and checking the global virtual address, this address is transfered along with any data in the case of puts to the target processing element. At the target element, the global virtual address is translated into a physical address in the cached region of the address space by the global translation buffer (GTB). This buffer supports several page sizes ranging from 64 KB to 128 MB and is completely managed in hardware, hence does not fault under normal conditions. Any access violations within the global segment are detected during this translation as well. Note that the translation at the target processing element allows each element to manage its own memory independently of other processing elements as long as the global translation buffer is updated accordingly. For example, the system control logic supports background copy operations to local memory without any impact on remote references.

In contrast to the earlier Cray T3D, cache coherence is maintained in hardware: If the remote memory access modifies the memory location, the external backmap is used to invalidate any cache-lines in the internal caches of the 21164 microprocessor as well as the stream buffers. If the remote memory access references a location that contains invalid data, the request is serviced from the valid data in the internal caches. Note that cache coherency is only

maintained within a single processing element, i.e. there is no global cache coherency protocol.

In the case of a read request, the requested data is transferred from the target to the source and is stored in the selected source-and-destination E-register, where the data can be retrieved by a simple load instruction. The accesses to the E-registers are synchronized by full/empty bits. A load that accesses an empty E-register, i.e. one that is used in an outstanding E-register operation, will stall until the operation is completed and the E-register contains the requested data.

The E-registers support a rich set of communication and synchronization commands:

- The get command returns the contents of a location in local or remote memory, i.e. performs a local or remote memory read.
- The put command updates the contents of a location in local or remote memory, i.e. performs a local or remote memory write.
- The swap command updates the contents of a location in local or remote memory similar to the put command, and returns the original contents of the local or remote memory location.
- The conditional swap command updates the contents of a location in local or remote memory, provided that the original contents meet the specified condition: If the original contents meet the condition, the local or remote memory location is updated and the original contents are returned. Otherwise, the local or remote memory location is unchanged and the original contents are returned.
- The fetch-and-increment command increments the contents of a location in local or remote memory and returns the original contents.
- The fetch-and-add command adds the specified value to the contents of a location in local or remote memory and returns the original contents.
- The send command stores a message to a local or remote memory location. The specified memory location must contain a message queue control word in order to ensure proper delivery of the message.
- The state command returns the contents of the specified state register.

These commands can be combined with several qualifiers, a detailed introduction to E-register programming is provided in Appendix B.

6.1.3 Network

The individual processor elements are connected by a three-dimensional torus network. Each processing element contains one network router that represents a node in the torus. In contrast to the earlier Cray T3D, there is only one processor instead of two per network node. Each router contains a full-duplex processor interface, two full-duplex network links for each dimension, as well as one I/O link. The network links are 14 bit (1 phit) wide and operate at 75 MHz using time-multiplexed transmission: In each clock cycle 5

phits (1 flit) are transmitted yielding a peak bandwidth of 600 MB/s. Due to protocol overhead, the maximum data bandwidth is reduced to approximately 500 MB/s. Note that the size of one flit is large enough to store a 64 bit quadword as well as some control information.

The network supports the individual E-register commands described above by providing single read- and write-requests, vector read- and write-requests, 8 quadword messages, atomic memory operations and configuration packets. The individual packets are between two and 10 flits long. The router chip supports five virtual channels [Dal90] for normal packets as well as an additional virtual channel for synchronization packets. Two virtual channels are used to remove turn cycles and request-response cycles, respectively, torus cycles are removed by routing packets in the first order of dimension. The fifth virtual channel provides a non-deterministic adaptive routing network, which does not remove any cycles. However, packets may reenter the deterministic routing network to avoid deadlock situations.

The router is based on a crossbar with additional paths for packets that do not turn, i.e. start routing in another dimension. Each network link provides buffers for all virtual channels, the buffers for the channels used in the deterministic network hold up to 12 flits, while the buffers for channels used in the non-deterministic network hold up to 22 flits. Upon entry into the first router, the logical number of the target processing element is used to index the routing lookup table. This table has 544 entries, each entry contains the physical address of the target node as well as the deterministic path to that node. For systems with more than 544 processing elements, i.e. 512 user and 32 support elements, the two least-significant bits are ignored, hence processors are mapped in groups of four, thus supporting up to 2176 processing elements, i.e 2048 user and 128 support.

Routing in the deterministic network is done in first order of dimension: Routing in positive direction of the x dimension (+x) comes first, followed by routing in the +y, +z, -x, -y, and -z dimensions. Note that the ordering of direction is fixed, but not the ordering of dimension, i.e. a packet with a -x, +y, +z routing tag would be routed in positive direction of the y and z dimensions first, afterwards in the negative direction of the x dimension. This routing scheme adds flexibility by providing multiple routes between two nodes, which is useful for fault tolerance.

The flexibility of the routing scheme is further enhanced by allowing initial and final hops: The first hop of a packet can be in positive x, y, or z dimension without changing the routing information in the packet itself. However, the final hop of a packet can only be in negative direction of the z dimension. Apart from adding flexibility to the routing scheme, initial and final hops enable the use of partially filled planes, thus reducing the minimum number of processor elements that have to be added in each step. Nodes in a partial plane without a direct route can be routed by using an initial hop in the +z

dimension as well as a final hop in the -z dimension, as all partial planes are oriented along the z dimension.

Adaptive routing allows requests and responses to be routed around local congestion in the network by allowing packets to turn in every node in a direction that brings them closer to their destination. Note that adaptive routing may cause reordering of packets and take broken links, hence adaptive routing can be turned off for certain packets and/or destinations. In addition, the adaptive virtual routing network is not guaranteed to be acyclic, hence packets must enter the deterministic, acyclic network in case of deadlocks. Note that the corresponding virtual channel has the lowest priority among all virtual channels, while the virtual channel for synchronization messages has the highest priority.

Instead of the dedicated synchronization network used in the earlier Cray T3D, the Cray T3E embeds the synchronization network into the normal network by using a high-priority virtual channel for synchronization packets. The system control logic on each processing element contains 32 barrier/eureka synchronization units. Each of these units provides a synchronization network for a specified set of processing elements. There are two different synchronization events: The barrier event forms a spanning tree across the processors using logical and, i.e. the event is propagated to the parent synchronization units as soon as all child units have signaled the event. The eureka event forms a spanning tree across the processors using logical or, i.e. the event is propagated to the parent synchronization units as soon as one of the child units has signaled the event. In addition, each router maintains a register for each of the 32 barrier/eureka synchronization units. The contents of the register configure the router as a node in the corresponding spanning tree by identifying the parent and child nodes. This information is used to propagate the synchronization events along the spanning tree.

6.1.4 Input/Output

In contrast to the earlier Cray T3D which handled all I/O requests via the Cray Y/MP foreground system, the Cray T3E provides its own set of I/O interfaces. As described in Section 6.1.3, each router has a dedicated bidirectional I/O link. The four processing elements on a printed circuit board share a common I/O control unit, which interfaces the corresponding I/O links to a GigaRing channel. Two GigaRing channels are shared by two I/O control units. The GigaRing channel is based on the SCI standard and provides a bidirectional link connected in a ring topology. The GigaRing provides a bandwidth of up to 500 MB/s in each direction, fault tolerance is provided by folding or masking the ring to isolate broken links: In the case of ring folding, the two neighbors of the broken node short circuit their internal paths to close the ring. In the case of ring masking, one direction in the ring is disabled. Note that the former does not affect the bandwidth of the ring, while the latter decreases the available bandwidth by a factor of two. The GigaRing

channels are used to connect the Cray T3E to other Cray computers and/or mass storage units. Fiber-optic cables can be used to extend the length of the GigaRing to 200 m.

6.1.5 Software

The Cray T3E is a self-hosted system that runs under the unicos/mk operating system, a distributed version of the unicos operating system used for the parallel vector processors from Cray Research. Both operating systems are derivates of the UNIX operating system. The unicos/mk operating system consists of a microkernel as well as several servers and provides a single system image. The microkernel is based on the CHORUS microkernel and is executed on every processing element. Note that the microkernel uses swapping instead of virtual memory and demand paging, i.e. the address space of a process is swapped entirely. There are several types of servers, the individual servers communicate via remote procedure calls and can be replicated for better performance.

The operating system distinguishes four different types of processing elements: command, application, operating system and redundant processing elements. Command processing elements are used to execute user commands and single-processor applications, while application processing elements are reserved for parallel applications. Operating system processing elements are reserved for the corresponding servers, while redundant processing elements are used to replace failed elements. The two former types are taken from the pool of user processing elements, the two latter types are taken from the pool of support processing elements.

Parallel applications can be executed in two different ways: interactive or batch. In interactive mode, the application is started from the command-line interface and executed as soon as a suitable partition of the machine is available. In batch mode, the application is started by submitting a corresponding request to the NQS scheduling system. The scheduling can by influenced by specifying several parameters, e.g. the maximum execution time.

The programming environment on the Cray T3E supports the following languages: CRAFT (Cray Research Adaptive Fortran), HPF (High-Performance Fortran), Fortran, C, and C++. CRAFT and HPF are extensions of the Fortran language targeted at efficient execution on distributed memory machines. Both Fortran 77 and 90 are supported as well as the standard C and C++ dialects. The individual compilers are augmented by the cld linker and the cam assembler, the totalview debugger supports the C and Fortran languages and allows source-level debugging of parallel applications.

The Message Passing Toolkit (MPT) contains the following communication libraries: PVM (Parallel Virtual Machine), MPI (Message Passing Interface), and shmem (Shared Memory). PVM [GBD+94] is a message passing library targeted at heterogeneous environments, MPI [Wal94] is the standard message-passing library used in commercial applications. The shmem

[Cra98b] library is proprietary and provides a set of routines for global memory accesses, collective operations and synchronization.

6.2 Methodology

Apart from the single-threaded base version, two versions using emulated multithreading were created for each benchmark: The bblk version uses basic blocks as instruction blocks, while the sblk version uses super block optimization to merge multiple basic blocks into one super block. All three versions of the benchmarks are derived from the same sources, although the bblk and sblk versions use the asynchronous communication routines from the emulation library instead of the synchronous communication routines from the shmem library. In addition, these two versions contain calls to the EMUthread_switch() routine in case of spin waits. Apart from these differences, all three versions of the benchmarks are built from the same sources. These sources are based on the original sources as distributed in the SPLASH2 benchmark suite, although the sources had to be modified during the porting process to the Cray T3E. The individual changes made to the sources during this process are described in Chapter 4.

The original sources were modified such that the performance counters are used for timing measurements instead of the operating system timers. The performance counter library (PCL) [BM98] from the Research Center Jülich is used to access the individual performance counters, especially the cycle counter. Note that the cycle counters are updated in each clock cycle, hence the resolution of the timing measurements is very high compared to the measurements using the operating system. Since all timing measurements are recorded as multiples of the processor cycle time, 64 bit integers have to be used in order to avoid overflows. Therefore the format of the timing statistics had to be changed to reflect the larger size of the results. All benchmarks were compiled with the -O2 optimization flag. This flag enables moderate automatic inlining, automatic scalar optimization and automatic vectorization.

All experiments were performed with three different problem sizes: the default problem size for the given benchmark as well as two and four times the default problem size. Experiments using different machine configurations ranging from 1 to 64 processors were performed for each of the benchmarks. In the case of emulated multithreading, the bblk and sblk versions of each benchmark were executed using different numbers of threads ranging from 1 to 16. In addition to the parallel experiments, the sequential experiments described in Chapter 5 were repeated on the Cray T3E in order to compare the sequential results on the two different platforms. The executables used during the sequential experiments are derived from the same sources that were used during the evaluation on the Compaq XP1000 workstation.

Recall from Section 5.10 that the sblk version is faster than all other versions using emulated multithreading for all benchmarks and problem sizes. This statement does not apply only to the Compaq XP1000 platform, but to the Cray T3E platform as well: For all sequential and parallel experiments and all problem sizes, the sblk version is always faster than the bblk version. Hence only the sblk results of the individual benchmarks will be presented in Sections 6.3 to 6.7. The sequential results were omitted as well, since they provide no additional insights compared to the experiments on the Compaq XP1000 platform.

The figures used to illustrate the results of the individual benchmarks are all structured in the same way: The horizontal axis reflects the number of processors, while the vertical axis represents the speedup relative to the runtime of the base version on a single processing element. In each figure, seven curves are used to illustrate the results: The circles represent the speedups for the base version, while the squares, diamonds, upward triangles, leftward triangles, upward triangles, and rightward triangles represent the speedups of the sblk version using 1,2,4,8, and 16 threads, respectively. Last but not least, the dots represent linear speedup and are provided to ease interpretation of the results. For each of the benchmarks three of these figures are provided, one for each problem size.

All experiments were performed in batch mode using the NQS queuing system in the Cray T3E, i.e. the corresponding executables had exclusive access to the processing elements. However, the NQS system assigns noncontiguous processing elements to one partition, i.e. the maximum number of routing steps between processing elements in such a partition is larger than the maximum number of routing steps required for a partition of the given size. These discontinuities may influence the benchmark results, since it is not possible to ensure that all experiments using a certain partition size are executed on partitions of the same structure. Unfortunately, there is no way to determine the structure of the partition for a given benchmark run. However, the number of routing steps is usually only one or two steps higher than the minimum number of routing steps, hence the variations in runtime should be rather small, especially for larger partitions. Discontinuous partitions can only be avoided by running the machine in dedicated mode which is not possible as the Cray T3Es in Jülich are used by researcher all over Germany.

6.3 FFT

The project-specific configuration file for the parallel version of the fft benchmark contains three internal and three external procedures, system and library routines are covered in the platform-specific configuration file. The SlaveStart() procedure is the entry point of the parallel algorithm and contains a call to the internal FFT1D() procedure apart from some initialization

and bookkeeping tasks. The FFT1D() procedure implements the six step algorithm described in Section 4.2.1 and contains several external calls as well as an internal call to the Transpose() procedure. The Transpose() procedure transposes the source matrix into the target matrix and is the only procedure that contains references to remote memory. The three procedures have to be internal for the following reasons: Apart from being the entry point, the SlaveStart() procedure contains two barrier synchronization as well as an internal call and therefore has to be internal. Due to the five barriers and three internal calls, the FFT1D() procedure has to be internal as well. Last but not least, the Transpose() procedure has to be internal as it contains several references to remote memory.

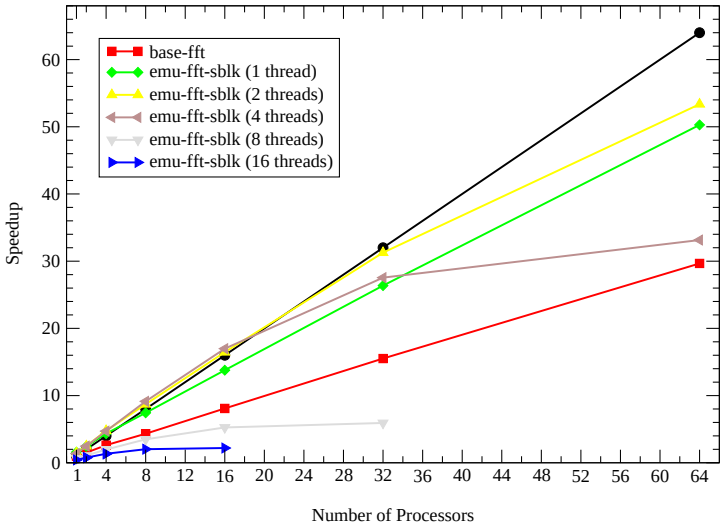
The three procedures consist of 21, 65, and 71 basic blocks, the assembler converter creates 7, 9, and 46 super blocks from these basic blocks. Note that the Transpose() procedure consists of more than one super block in contrast to the results on the Compaq XP1000 platform, which is explained as follows: In the parallel version of the fft benchmark, the Transpose() procedure contains references to remote memory, i.e. calls to the communication routines of the emulation library. These calls force the end of a super block such that a context switch is performed right after the remote memory access has been initiated.

The results of the experiments using the fft benchmark are summarized in Figures 6.3, 6.4, and 6.5. Figure 6.3 illustrates the results for a problem size of 64 K points, while Figures 6.4 and 6.5 illustrate the results using problem sizes of 256 K and 1024 K points, respectively. All three figures have an identical structure as described in Section 6.2.

Using a problem size of 64 K points, the source and target matrices occupy approximately 1 MB of memory each. Recall that the fft benchmark aligns each row of these matrices on cache-line and pagesize boundaries, hence the matrices are slightly larger than the 1 MB that is required to store the actual data. There are three different matrices of this size: two matrices are used as source and target during transpose operations, the third matrix contains the roots-of-unity. In addition, the first row of the latter matrix is replicated by each thread on all processors.

According to [WOT+95] the first- and second-level working sets for the fft benchmarks are one row of the matrix and one partition of the matrix, respectively. The size of the first-level working set is independent of the number of processors and is 4 KB. As the 21164 microprocessor used in the Cray T3E contains an 8 KB first-level data cache and a 96 KB second-level unified cache, it is likely that the first-level working set will reside in one of the internal caches. The size of the second-level working set is approximately $3\,\mathrm{MB}/p$ for all processors, where p is the number of processors. Although the working sets for the individual threads on a processor are even smaller, the working set of all threads on a processor will overlap due to multithreaded execution.

Fig. 6.3. Results for the FFT Benchmark (64 K Complex Data Points)



An analysis of the data presented in Figure 6.3 yields the following results: The sblk version of the fft benchmark is faster than the base version as long as no more than four threads are used. However, the advantage of the sblk version decreases with growing number of processors, especially for the sblk version using four threads. This behavior is probably caused by the small working set for each thread and processor: Using 64 processors and 4 threads, each processor has a working set of less than 48 KB, i.e. 12 KB for each thread. This reasoning is supported by the fact that the speedups deteriorate significantly slower if one of the larger problem sizes is used. The sblk version is slower than the base version if more than four threads are used. This can be explained by a combination of small working sets per threads and the increase in runtime due to the larger number of threads. Compared to the Compaq XP1000 platform, this effect is more pronounced on the Cray T3E as a context switch is executed in the inner loop of the three loop nests in the Transpose() procedure. In addition, the larger number of thread descriptors increases the number of cache misses as the first-level cache used in the 21164 processor is only 8 KB compared to the 64 KB of the 21264. Note that the results using eight or 16 threads are not complete as the problem size is too small for 1024 threads.

Using a problem size of 256 K points, each of the three matrices occupies approximately 4 MB of memory for a total of 12 MB. The sizes of the first- and second-level working sets increase to 8 KB and 12 MB/p, respectively. Note that the first-level working set is now the same size as the first-level data cache. An analysis of the data presented in Figure 6.4 yields the same results as for the smaller problem size, although all versions using emulated multithreading perform significantly better than for the smaller problem size.

Fig. 6.4. Results for the FFT Benchmark (256 K Complex Data Points)

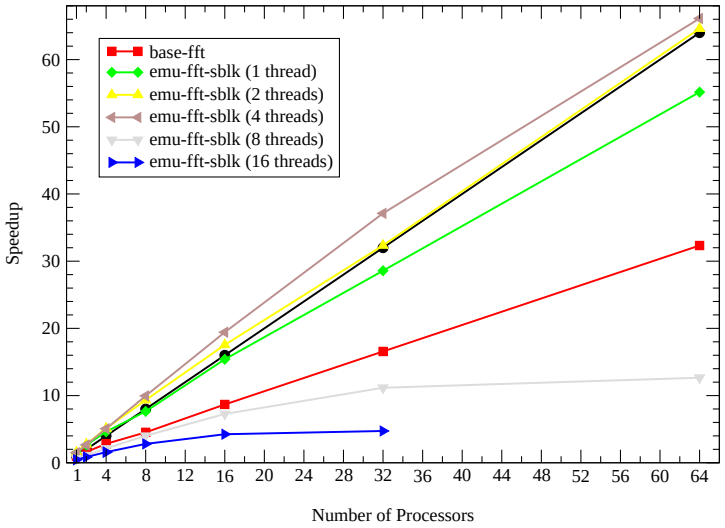
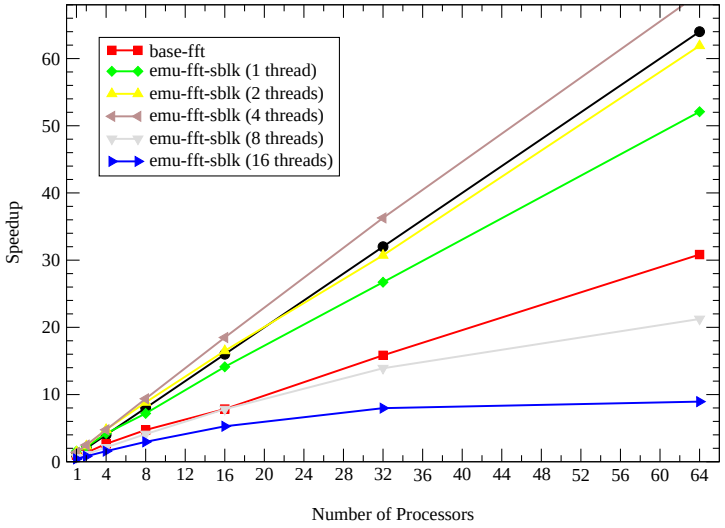


Fig. 6.5. Results for the FFT Benchmark (1024 K Complex Data Points)



Although some of the results seem to indicate super-linear speedup, this is not the case: If the sblk version on a single processor is faster than the base version on a single processor, this phenomenon is caused by the fact that all speedups are relative to the base version on a single processor. The result for the sblk version using 16 threads on 64 processors is missing as the problem size is too small for 2048 threads.

Using a problem size of 1024 K points, each of the three matrices occupies approximately 16 MB of memory for a total of 48 MB. The sizes of the first- and second-level working sets increase to 16 KB and 48 MB/p, respectively. Note that the first-level working set is now larger than the first-level data cache. An analysis of the data presented in Figure 6.5 yields the same results as for the smaller problem size, although the sblk version using up to two or four threads deteriorate slightly compared to the 256 K problem size. As the sblk version using one thread is not affected, this is probably caused by an increased number of cache misses.

In summary, the results using the sblk version of the fft benchmark are quite encouraging: As long as no more than four threads are used, the sblk version is significantly faster than the base version, especially for the two larger problem sizes. However, the sblk version using eight or more threads is always slower than the base version, probably due to the increased overhead and the smaller working set per processor/thread.

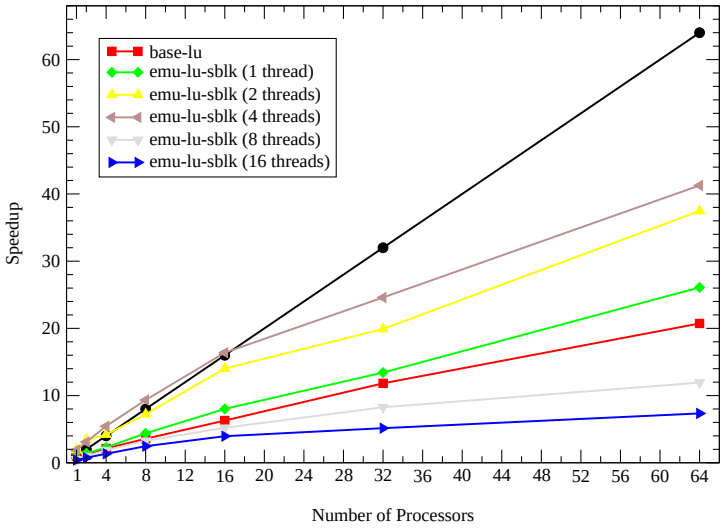
6.4 LU

The project-specific configuration file for the parallel version of the lu benchmark contains six internal and four external procedures, system and library routines are covered in the platform-specific configuration file. Note that the number of internal procedures is larger than for the sequential version used on the Compaq XP1000 platform: The SlaveStart(), OneSolve() and lu() procedures common in both versions have to be internal since they contain barrier synchronizations or calls to other internal procedures. The remaining three procedures contain references to remote memory, i.e. calls to the communication routines of the emulation library. In order to hide the latency of these references, a context switch is performed right after the reference has been initiated, i.e. these calls force the end of a super block.

The six internal procedures consist of 177 basic blocks, the assembler converter constructs 67 super blocks by merging multiple basic blocks into one super block. Note that the parallel version of the lu benchmark on the Cray T3E uses a larger number of basic blocks than the sequential version of the lu benchmark on the Compaq XP1000 platform. This difference is caused by the additional internal procedures and the different number and structure of basic blocks due to different compiler technology, as none of the three common internal procedures contains references to remote memory that could limit the size of the super blocks.

The results of the experiments using the lu benchmark are summarized in Figures 6.6, 6.7, and 6.8. Figure 6.6 illustrates the results using a matrix size of 512×512, while Figures 6.7 and 6.8 illustrate the results using matrix sizes of 1024×1024 and 2048×2048, respectively. All three figures have an identical structure as described in Section 6.2.

Fig. 6.6. Results for the LU Benchmark (512 × 512 Matrix)



Using a matrix size of 512 × 512, the corresponding matrix occupies 2 MB of memory. The matrix is partitioned into blocks of size 16 × 16, hence each block occupies 2 KB of memory. According to [WOT+95], the first- and second-level working sets for the lu benchmark are one block of the matrix and one partition of the whole data set, respectively. The first-level working set fits in the 8 KB first-level data cache of the 21164 processor, while the second-level working set might fit into the 96 KB second-level cache for larger numbers of processors.

An analysis of the data presented in Figure 6.6 yields the following results: The base version does not scale well, probably due to the small problem size. The sblk version of the lu benchmark is always faster than the base version as long as no more than four threads are used. The sblk version using eight or more threads is significantly slower, reflecting the increased overhead with a growing number of threads. However, the sblk version using four threads is the fastest, striking a good balance between the overhead by additional threads and the ability to tolerate the latency of remote memory references. Similar to the base version, the speedups of the sblk version deteriorate somewhat if more than 16 processors are used, probably due to the small problem size.

Using a matrix size of 1024 × 1024, the corresponding matrix occupies 8 MB of memory, while the individual blocks are still 2 KB large. Hence the first-level working set will still fit in the 8 KB first-level data cache, while the second-level working set will no longer fit in the second-level cache even for the largest number of processors. An analysis of the data presented in Figure 6.7 yields the same results as for the smaller problem size, although the performance of the sblk version is even better: The speedups of the sblk version deteriorate only if 64 processors are used. This evidence supports the

Fig. 6.7. Results for the LU Benchmark (1024 × 1024 Matrix)

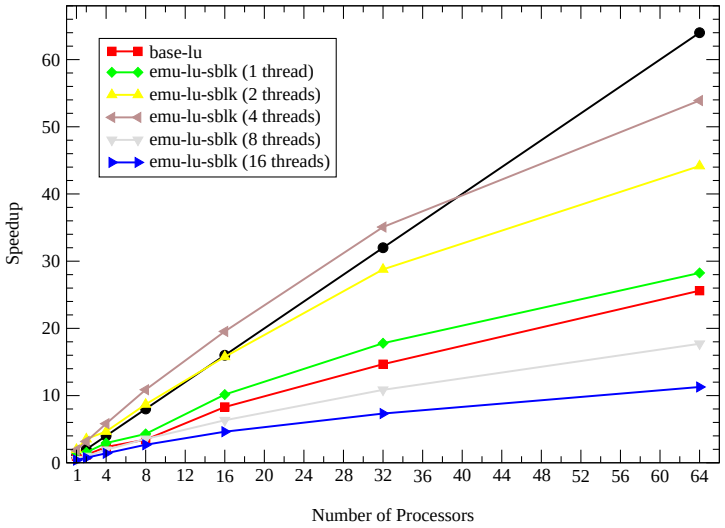
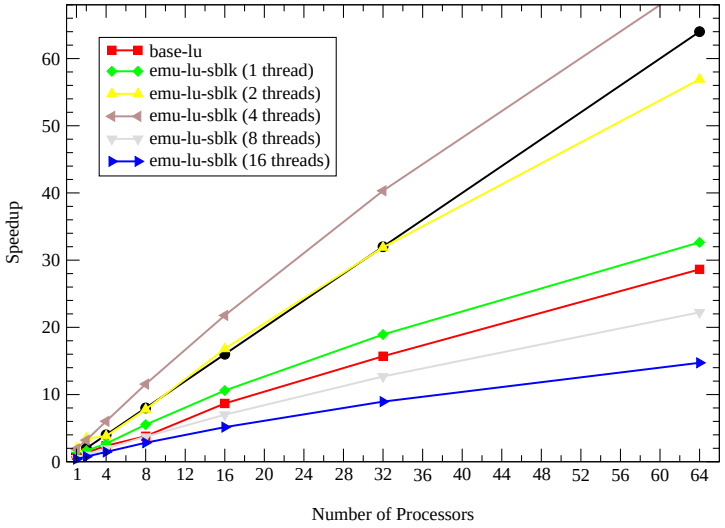


Fig. 6.8. Results for the LU Benchmark (2048 × 2048 Matrix)



fact that the deterioration for the smaller problem size was indeed caused by the problem size.

Using a matrix size of 2048 × 2048, the corresponding matrix occupies 32 MB of memory, while the individual blocks are still 2 KB large. Like before, the first-level working set will still fit in the 8 KB first-level data cache, while the second-level working set will not fit in any of the internal caches. An analysis of the data presented in Figure 6.8 yields the same results as for

the smaller problem sizes, although the performance of the sblk and base versions is even better, i.e. the runtimes of both versions scale well up to 64 processors.

In summary, the results for the lu benchmark are quite encouraging: As long as no more than four threads are used, the sblk version is always significantly faster than the base version. The sblk version using four threads seems to strike a good balance between the overhead due to additional threads and the ability to hide the latency of remote memory references.

6.5 Radix

The project-specific configuration file for the parallel version of the radix benchmark contains one internal and three external procedures, system and library routines are covered in the platform-specific configuration file. The internal slavesort() procedure is the entry point of the parallel algorithm and implements the radix sort algorithm described in Section 4.2.3, the three external procedures are only used during initialization of the sort array. Apart from being the entry point, the slavesort() procedure has to be internal as it contains barrier synchronizations, spin waits, as well as references to remote memory.

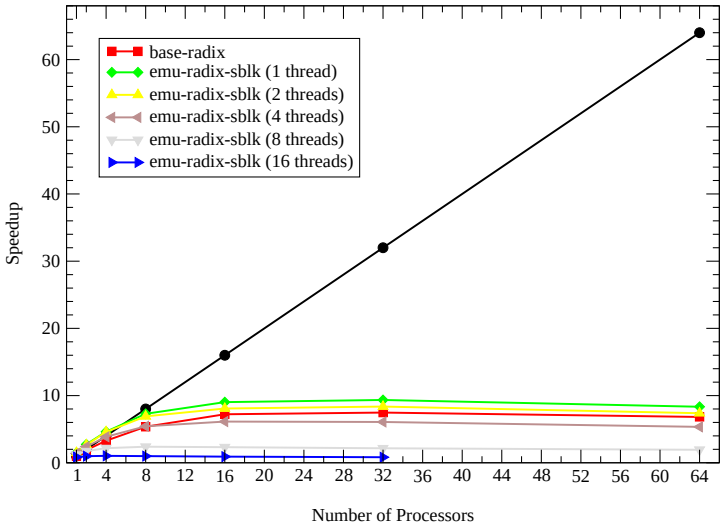
The slavesort() procedure consists of 139 basic blocks and 44 super blocks. Note that the number of super blocks on the Cray T3E platform is higher than the number of super blocks in the sequential version of the radix benchmark used on the Compaq XP1000 platform. Apart from different compiler technology, i.e. a different number and structure of basic blocks, this is caused by references to remote memory, i.e. calls to the communication routines of the emulation library. These calls force the end of a super block such that a context switch is performed right after the remote memory access has been initiated.

The results of the experiments using the radix benchmark are summarized in Figures 6.9, 6.10, and 6.11. Figure 6.9 illustrates the results using a problem size of 256 K integers, while Figures 6.10 and 6.11 illustrate the results using problem sizes of 512 K and 1024 K integers, respectively. All three figures have an identical structure as described in Section 6.2.

Using a problem size of 256 K integers, the sort array occupies 2 MB of memory, since the Cray T3E uses 64 bit integers by default. As radix sort does not sort in place, two of these arrays are needed, thereby occupying 4 MB of memory. In addition, each thread maintains a histogram of the local keys, the size of the corresponding arrays depends on the selected radix. Using a radix of 1024, each of these arrays occupies 8 KB in memory. Note that the radix is independent of the problem size, i.e. the same radix was used for all experiments.

According to [WOT+95], the first- and second-level working sets for the radix benchmark are a histogram and one partition of the whole data set,

Fig. 6.9. Results for the Radix Benchmark (256 K Integers)



respectively. The size of the first-level working set is independent of the number of processors and threads and is 8 KB large. As the 21164 microprocessor used in the Cray T3E contains an 8 KB first-level and a 96 KB second-level cache, the first-level working set might not fit completely into the first-level data cache. The size of the second-level working set is $4\,\mathrm{MB}/p$ for each processor as the working sets of all threads on a given processor are likely to overlap due to multithreaded execution.

An analysis of the data presented in Figure 6.9 yields the following results: The base version does not scale well with the number of processors, at least for the given problem size. This is probably caused by the parallel prefix operations used to collect and distribute the global rank arrays, which are not completely parallelizable: The radix benchmark uses an array of $2p$ prefix nodes to build a binary tree across all processors. Each of the prefix nodes represents a node at a certain level in the tree and is mapped to a processor based on the processor number. Although more than one prefix node is mapped to the same processor, all prefix nodes mapped to the same processor are from different levels in the tree. Note that only one processor is active in all levels of the tree. During the gather operation, each processor copies its local rank array to the corresponding prefix node in the lowest level. Afterwards all processors, that are active at a given level, wait until both children of the corresponding prefix nodes have been updated and combine these rank arrays. The result of the combination is stored in the current prefix node. The scatter operation is similar to the gather operation, but works in reverse order. These operations do not scale well as most of the processors are idle, especially in the upper levels of the tree. However, the Cray T3E supports gather and scatter operations in hardware via the embedded synchronization

Fig. 6.10. Results for the Radix Benchmark (512 K Integers)

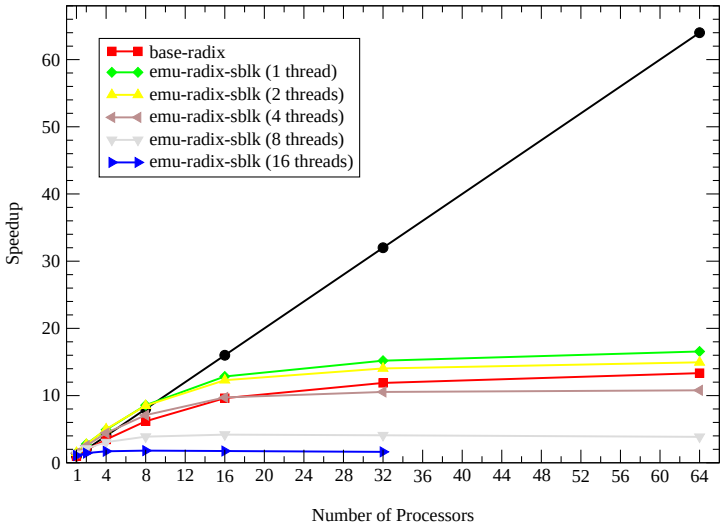
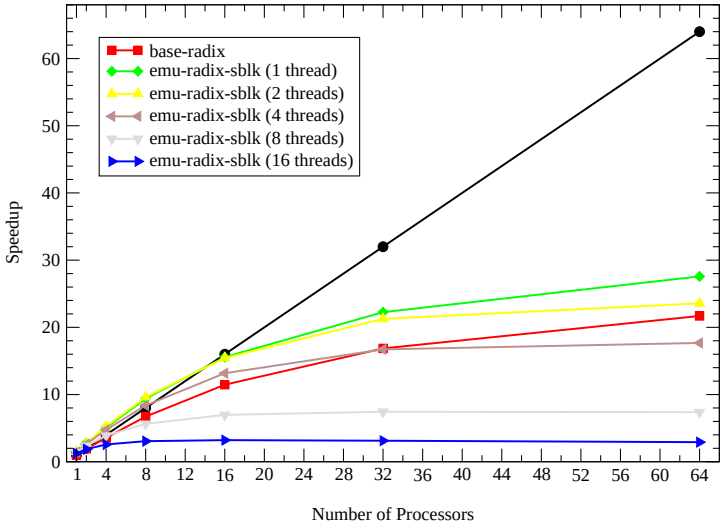


Fig. 6.11. Results for the Radix Benchmark (1024 K Integers)



network. Using the corresponding gather and scatter operations provided by the shmem library should improve the performance of the distribution and broadcast of the local ranks significantly.

The sblk version of the radix benchmark is slightly faster than the base version across all numbers of processors, as long as no more than two threads are used. The sblk version using four threads is slightly slower than the base version, at least on more than eight processors. This behavior is probably due

to the small working sets for each thread/processor as well as an increased overhead using more threads. Recall that the overhead on the Cray T3E is larger than the overhead on the Compaq XP1000 due to the smaller size of the instruction blocks. The effect is even more pronounced for the sblk version using eight or 16 threads, which do not seem to benefit from additional processors at all. The above reasoning is supported by the fact that all sblk versions scale better in the case of the two larger problem sizes.

Using an input size of 512 K integers, each of the two sort arrays occupies 8 MB of memory for a total of 16 MB. The size of the local histogram maintained by each thread is independent of the problem size, the corresponding arrays still occupy 8 KB of memory. Compared to the smaller problem size, the size of the first-level working set, i.e. a histogram, is the same, while the size of the second-level working set, i.e. a partition of the whole data set, doubles to 16 MB/p. An analysis of the data presented in Figure 6.10 yields the same results as for the smaller problem size, although all versions of the radix benchmark scale slightly better. This is probably due to the increased size of the working sets for each thread/processor.

Using an input size of 1024 K integers, each of the two sort arrays occupies 16 MB of memory for a total of 32 MB. Again, the size of the local histogram maintained by each thread is independent of the problem size, the corresponding arrays still occupy 8 KB of memory each. Compared to the smallest problem size, the size of the first-level working set, i.e. a histogram, is the same, while the size of the second-level working set, i.e. a partition of the whole data set, quadruples to 32 MB/p. An analysis of the data presented in Figure 6.11 yields the same results as for the two smaller problem sizes, although all versions of the radix benchmark scale even better than before. This fact provides further evidence that the increased working sets for each thread/processor is responsible for this behavior.

In summary, the results for the radix benchmark are quite encouraging, since the sblk version is always faster than the base version as long as no more than four threads are used. Neither the base nor the sblk versions scale well with an increased number of processors. However, the focus of these experiments is the evaluation of emulated multithreading, the performance of the base versions is not the primary concern. As outlined above, the performance of the radix benchmark could probably be improved by using the scatter and gather operations provided by the shmem library instead of the hand-coded version found in the original sources.

6.6 Ocean

The project-specific configuration file for the parallel version of the ocean benchmark contains eleven internal and three external procedures, system and library routines are covered in the platform-specific configuration file. Note that the number of internal procedures increases from three for the

sequential version of the ocean benchmark to 11 for the parallel version. The three slave(), slave2(), multig() procedures continue to be internal, since they contain synchronization points or calls to the remaining procedures. The remaining eight procedures contain references to remote memory, i.e. calls to the communication routines of the emulation library. Hiding the latency of these references is only possible if a context switch is performed right after the remote memory access has been initiated, hence the corresponding procedures should be internal. The 11 internal procedures consist of 1892 basic blocks and 523 super blocks.

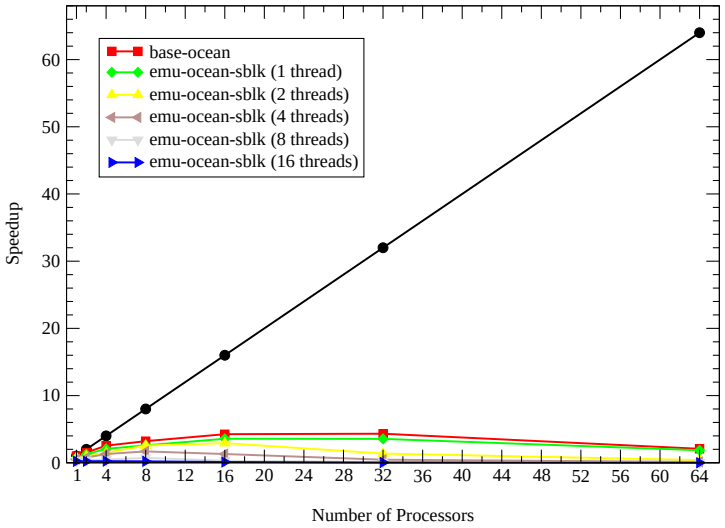
The results of the experiments using the ocean benchmark are summarized in Figures 6.12, 6.13, and 6.14. Figure 6.12 illustrates the results using an ocean with 130×130 grid points, while Figures 6.13 and 6.14 illustrate the results using oceans with 258×258 and 514×514 grid points, respectively. All three figures have an identical structure as described in Section 6.2.

Using a grid of 130×130 points, one of the corresponding arrays occupies approximately 132 KB of memory. Recall that the ocean benchmark uses 25 of these arrays for a total size of approximately 3 MB. Apart from these and several smaller arrays, the ocean benchmark uses two arrays as input to the multigrid solver. Each of these arrays has several levels, the number of levels is equal to the binary logarithm of the problem size. For the given problem size, each of the arrays has seven levels, thereby both arrays occupy approximately 2 MB of memory.

According to [WOT+95], the first- and second-level working set consists of a few subrows of one of the matrices and a partition of the whole data set, respectively. The first-level working set occupies a few KB, while the second-level working set occupies approximately $5\,\mathrm{MB}/p$ due to the multithreaded execution. While the first-level working set will probably fit in the 8 KB first-level data cache, the second-level working set will not fit in any of the internal caches unless 64 or more processors are used.

An analysis of the data presented in Figure 6.12 yields the following results: The base version of the radix benchmark does not scale well, most of the time is spent in the multigrid solver. In particular, the time spent in the multigrid solver increases significantly with a growing number of processors. It is unlikely that this behavior is caused by the small working sets alone, as good speedups have been reported for the ocean benchmark on other platforms. However, the ocean benchmark scaled even worse on previous runs of the experiments, this was traced back to a bug in the assembler: Although the compiler creates correct code, the assembler miscalculated the offset of a critical constant such that unpredictable values were used. This issue could be resolved by renaming the constant, yielding a factor 32 (!) speedup on 64 processors. Due to computing time constraints, it was not possible to determine whether the poor performance of the ocean benchmark is caused by other bugs of the same kind.

Fig. 6.12. Results for the Ocean Benchmark (130 × 130 Ocean)



Unfortunately, the performance of the sblk version is always worse than the performance of the base version, although the version using one thread is close. This indicates that the performance degradation is due to the overhead caused by context switches: The ocean benchmark contains a large number of loops and most of these loops contain references to remote memory, i.e. a context switch is performed for every iteration of these loops.

Using a grid size of 258 × 258, a corresponding array occupies approximately 520 KB of memory, i.e. a total of 13 MB for the 25 arrays in the ocean benchmark. The two arrays used as input to the multigrid solver occupy approximately 7 MB, as they are now eight levels deep. The first-level working set has doubled in size and is probably too large to fit in the 8 KB first-level data cache of the 21164 microprocessor. However, the working set will still fit in the 96 KB second-level cache. The second-level working set occupies approximately 20 MB/p, where p is the number of processors. Even for the largest configuration, i.e. 64 processors, the second-level working set will not fit in any of the caches.

An analysis of the data presented in Figure 6.13 yields the same results as for the smaller problem size, although all versions of the ocean benchmark scale better than before. However, the sblk version is always slower than the base version, independent of the number of threads. Note that the distance between the base and sblk versions increases as well, probably due to the increased number of loop traversals for the larger problem size.

Using a grid size of 514 × 514, a corresponding array occupies approximately 2 MB of memory, i.e. a total of 50 MB for the 25 arrays in the ocean benchmark. The two arrays used as input to the multigrid solver occupy approximately 32 MB, as they are now nine levels deep. The first-level working

Fig. 6.13. Results for the Ocean Benchmark (258 × 258 Ocean)

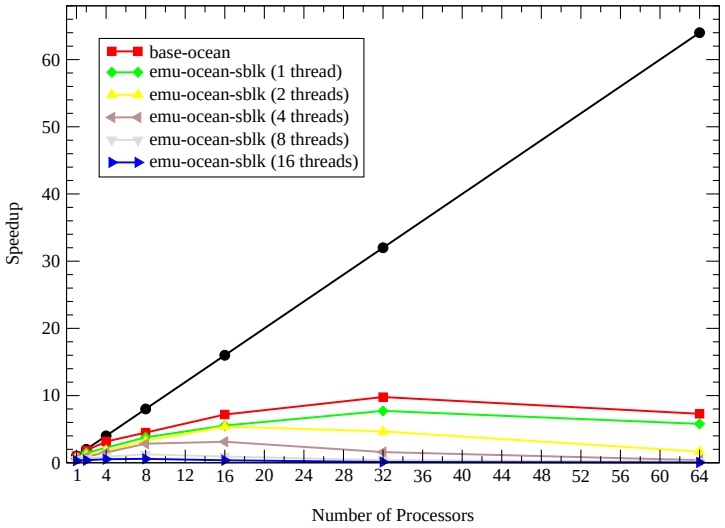
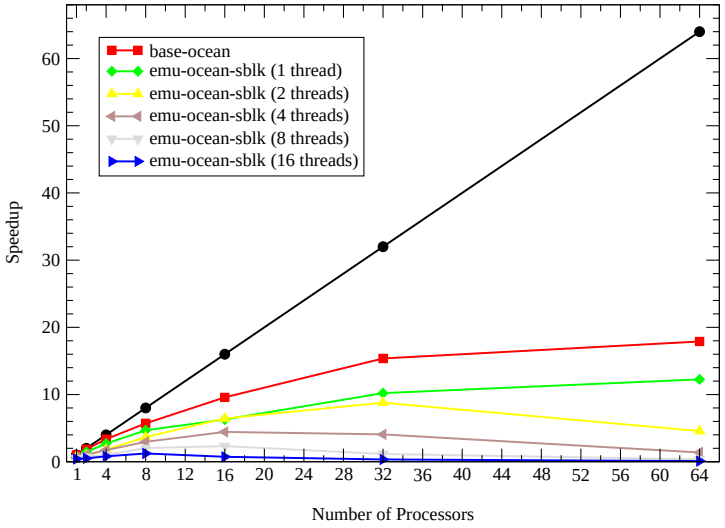


Fig. 6.14. Results for the Ocean Benchmark (514 × 514 Ocean)



set has doubled in size again, occupying some tens of KB and is too large to fit in the 8 KB first-level data cache of the 21164 microprocessor. However, the working set may still fit in the 96 KB second-level cache. The second-level working set occupies approximately 82 MB/p, where p is the number of processors. An analysis of the data presented in Figure 6.14 yields the same results as for the smaller problem size, although all versions of the ocean benchmark scale better than before. However, the sblk version is al-

ways slower than the base version, independent of the number of threads. The distance between the base and sblk versions increases as well, probably due to the increased number of loop traversals for the larger problem size.

In summary, the results for the ocean benchmark are disappointing: For all problem sizes, the sblk version is always slower than the base version. In addition, the distance between the two versions increases with growing problem size. This is probably caused by the overhead due to context switches, since the ocean benchmark contains a large number of loops that contain remote memory references. This problem can be solved by improving the efficiency of the main loop in the thread execution routine, the use of static prediction to determine whether a given remote memory reference should force the end of a super block. In addition, register partitioning could be used to decrease the context switch overhead, especially if a small number of threads is used.

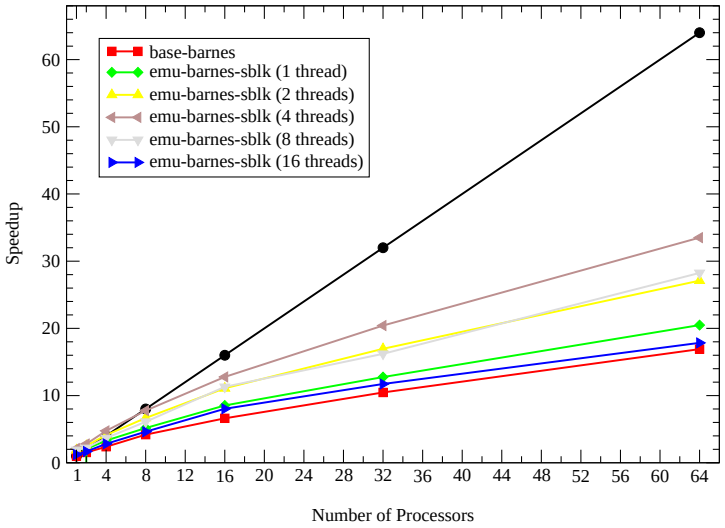
6.7 Barnes

The project-specific configuration file for the parallel version of the barnes benchmark contains 13 internal and six external procedures, system and library routines are covered in the platform-specific configuration file. Note that the number of internal procedures increases from five for the sequential version to 13 for the parallel version. The five common internal procedures still have to be internal since they contain synchronization points or calls to other internal procedures. The remaining eight procedures contain references to remote memory, i.e. calls to the communication routines of the emulation library. Hiding the latency of these references requires that a context switch is performed right after the remote memory access has been initiated, hence the corresponding procedures should be internal. The 13 internal procedures in the ocean benchmark consist of 425 basic and 261 super blocks.

The results of the experiments using the barnes benchmark are summarized in Figures 6.15, 6.16, and 6.17. Figure 6.15 illustrates the results using a problem size of 16 K particles, while Figures 6.16 and 6.17 illustrate the results using problem sizes of 64 K and 256 K particles, respectively. All three figures have an identical structure as described in Section 6.2.

Using a problem size of 16 K particles, approximately 2 MB of memory is used for the particle array. In addition, 34 MB and 20 MB of memory is used for the local cell and leaf arrays as well as approximately 16 MB for the local cell and leaf pointer arrays. According to [WOT+95] the first-level working set is the tree data for one particle, i.e. the leaf that contains the particles as well as all cells on the path from the root cell to that leaf. As the height of the tree is logarithmic in the number of particles and the size of a leaf and cell is 264 and 216 bytes, the first-level working set is a few KB large. Note that the size of the leaf and cell structures on the Cray T3E is different from the size of these structures on the Compaq XP1000 since the Cray T3E uses 64 bit integers by default. The second-level working set is a partition of

Fig. 6.15. Results for the Barnes Benchmark (16 K Particles)



the whole data set. While the first-level working set will probably fit in the first-level data cache, the second-level working set occupies 72 MB/p and will therefore not fit in any of the internal caches.

An analysis of the data presented in Figure 6.15 yields the following results: The base version does not scale well, even when the small problem size is taken into account. A detailed analysis of the runtime components reveals that most of the time is spent in the force computation phase. Due to time and resource constraints, it was not possible to determine whether the barnes benchmark is affected by similar bugs like the ocean benchmark. However, in contrast to the ocean benchmark, all sblk versions are faster than the base version.

Using a problem size of 64 K particles, 8.75 MB of memory is used for the particle array. In addition, 135 MB and 80 MB of memory is used for the local cell and leaf arrays as well as 66 MB for the local cell and leaf pointer arrays. Compared to the smallest problem size, the size of the first-level working set increases slightly, while the size of the second-level working set quadruples. While the first-level working set will probably still fit in the first-level data cache, the second-level working set occupies 290 MB/p and will therefore not fit in any of the internal caches. An analysis of the data presented in Figure 6.16 yields the same results as for the smaller problem size, although all versions of the barnes benchmark scale better than for the smaller problem size. The sblk versions are still faster than the base version for all numbers of threads and processors.

Using a problem size of 256 K particles, 35 MB of memory is used for the particle array. In addition, 540 MB and 320 MB of memory is used for the local call and leaf arrays as well as 264 MB for the local cell and leaf

Fig. 6.16. Results for the Barnes Benchmark (64 K Particles)

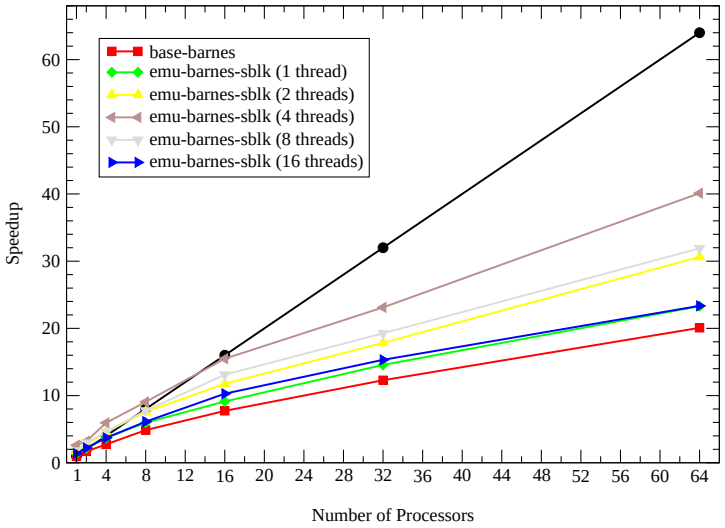
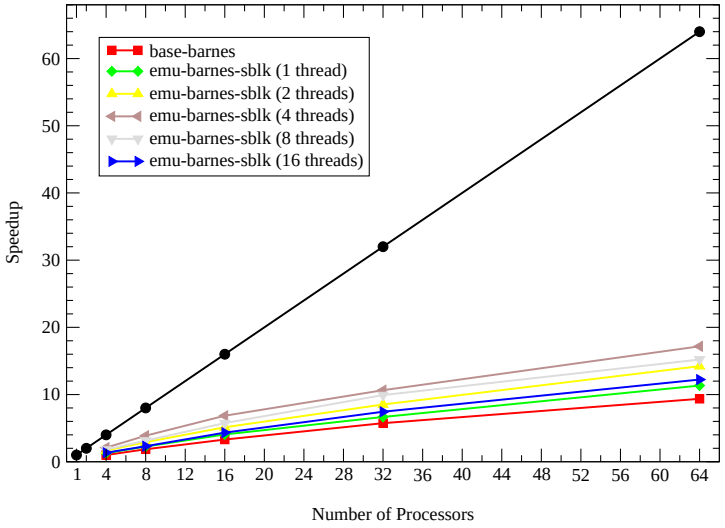


Fig. 6.17. Results for the Barnes Benchmark (256 K Particles)



pointer arrays. Compared to the smallest problem size, the size of the first-level working set increases slightly, while the size of the second-level working set increases by a factor of 16. While the first-level working set will probably fit in the first-level data cache, the second-level working set occupies 1159 MB/p and will therefore not fit in any of the internal caches. An analysis of the data presented in Figure 6.17 yields the same results as for the smaller problem sizes, although all versions of the barnes benchmark seem to

scale significantly worse than before. However, the results in Figure 6.17 use the base version on four processors as the baseline, i.e. the results can not be compared directly with the results from the smaller problem sets. Note that the barnes benchmark cannot be executed on less than four processors due to the large amount of memory required for the given problem size.

In summary, the results of the barnes benchmark are quite encouraging: For all numbers of threads and processors, the sblk version is faster than the base version, especially for a small number of threads. This result could probably be improved by using register partition in order to decrease the overhead associated with emulated multithreading.

6.8 Summary

Emulated multithreading is designed to tolerate the latency of remote memory references in massively parallel processors such as the Cray T3E. The evaluation of emulated multithreading on this platform has demonstrated that emulated multithreading is feasible: Apart from the ocean benchmark, some configuration of the sblk version is always faster than the base version, for all numbers of processors. This result is encouraging, especially as the 21164 processor used in the Cray T3E is not particularly well suited for emulated multithreading due to the small size of the first-level caches and the in-order instruction issue. In addition, the overhead on the Cray T3E is higher than the overhead on the Compaq XP1000, as the remote memory accesses limit the size of the super blocks. This problem can be addressed by using profiling information to decide whether the latency of a given remote memory access should be tolerated, i.e. the algorithm that creates the super blocks has to strike a balance between latency tolerance and overhead. Alternatively, register partitioning could be used to reduce the overhead associated with context switches.

Apart from the performance of the executables using emulated multithreading, the stability of these executables is better than expected: Although emulated multithreading changes the assembler code in a complex conversion process, all executables are stable. Note that the fft, lu, and radix benchmark use automatic self-checks to ensure the validity of the results, while the results for the ocean benchmark were checked manually. No discrepancies between the results of the base and the sblk version were found, indicating that the current implementation of emulated multithreading is quite mature.

7. Conclusions

The current chapter summarizes this work by providing a brief overview over the previous chapters. Afterwards the open questions and future developments regarding emulated multithreading are addressed.

The first chapter contains a detailed analysis of current trends in sequential and parallel computing based on the characteristics of past and present microprocessors. The analysis identified the characteristics of the memory system, i.e. bandwidth and latency, as one of the major bottlenecks that limit the performance of current microprocessors. On the one hand, bandwidth is less of a problem, since several techniques to increase the bandwidth are readily available, although the corresponding costs may be prohibitive. On the other hand, the latency of the memory system cannot be reduced below the inherent latency of the memory devices. Unfortunately, the gap between the clock frequency of microprocessors and the latency of the memory devices is steadily increasing, i.e. the latency as seen by the microprocessor in terms of clock cycles increases even faster. In the case of parallel computing, the situation is even worse due to the latency of the network that connects the individual processing elements. For references to remote memory, this latency has to be added to the latency of the local memory system, yielding latencies on the order of thousands of cycles.

After identifying the latency of the memory system as one of the major performance limitations for sequential and parallel computers, several techniques for tolerating instead of reducing latency were presented. Multithreading tolerates the latency by executing several threads of control on the same processor, such that other threads are executed while some threads wait for the completion of outstanding memory references. Multithreading is the most general of the latency tolerance techniques in the sense that it makes the least assumptions about the structure of the application. Therefore a large number of hardware and software approaches to multithreading were described in order to determine whether these approaches can be used to solve the problem mentioned above. However, multithreading has yet to find its way into mainstream microprocessors, i.e. multithreading has to be implemented in software on these microprocessors. Although there is a wide range of software multithreading systems, none was specifically designed to

tolerate the latency of local or remote memory references. Hence a new technique called emulated multithreading was designed for this purpose.

Based on the analysis in the first chapter, the second chapter motivated the fundamental design choices behind emulated multithreading using a well-known model of multithreaded execution. These choices enable emulated multithreading to support sufficiently small grain sizes that are needed to tolerate the latency of local or remote memory references: Emulated multithreading uses a static context switch strategy, i.e. context is switched after each initiation of a remote memory request, thereby hiding the latency of this request by executing other threads instead. The focus on remote memory references was chosen since these events incur a long latency and are easily identified. In order to reduce the context switch overhead, first-come, first-served scheduling was chosen for emulated multithreading. In addition, the context switch code is tailored to each context switch location and embedded into the application, thereby reducing the number of save and restore operations. Based on these design choices, the basic concept of the implementation of emulated multithreading is described, e.g. the data structures, support routines, as well as a the conversion process.

The individual elements of emulated multithreading were investigated with respect to performance issues in connection with the characteristics of current microprocessors, especially caches, branch prediction, scheduling, and out-of-order execution. This investigation revealed that emulated multithreading benefits from large first-level data and instruction caches, the ability to prefetch memory locations, branch prediction, and out-of-order execution. Based on these results, several current RISC and CISC architectures were examined in order to determine their suitability with respect to emulated multithreading. The Alpha architecture was chosen as the platform for the first implementation of emulated multithreading due to the clear architecture state, the support for prefetching, synchronization, branch hints, as well as the existence of out-of-order implementations and massively parallel processors based on this architecture.

The third chapter provided a detailed description of the implementation of emulated multithreading that consists of two separate converters as well as a small library. The high-level language converter is responsible for all code modifications in the high-level language sources as well as the creation of configuration files. However, the high-level language converter has not been implemented since integration into the frontend of a compiler has been favored. The assembler converter performs all code modifications on the assembler language sources, i.e. creates the tailored context switch code and embeds it into the application code. The assembler converter is quite complex and uses several techniques known from compiler design: Basic block creation partitions the sequence of assembler instructions into basic blocks, i.e. maximal sequential sequences. The corresponding algorithm was described in detail and a proof for the worst-case runtime was given. While basic block creation

is a standard technique used in every compiler, the assembler converter supports the creation of super blocks as well: A super block is a collection of basic blocks with a single entry point and multiple exit points. A survey of similar structures revealed that these structures were too limited for the purposes of emulated multithreading. As the evaluation of emulated multithreading has demonstrated, super blocks are an important optimization for emulated multithreading. Hence the corresponding algorithm was described in detail and proofs for the worst-case runtime and several important properties of super blocks were proven.

Data-flow analysis is an integral part of the assembler converter and a prerequisite for the register allocation phase. The assembler converter uses several different data-flow analyses: Apart from the traditional live and reaching analysis that were adapted to suit the needs of the assembler converter, two new data-flow analyses used to detect the start and end of the individual live ranges were invented. Using lattice theory, all four data-flow analyses were proven to form distributive data-flow analysis frameworks. Based on these results, the well-known iterative data-flow algorithm was used to compute the desired meet-over-all-paths solution for the for data-flow analyses.

The register allocator is probably the most complex part of the assembler converter and is used to reallocate all registers in the original source code. The allocation uses a heuristic approach based on graph coloring and reuses techniques from the two major approaches to register allocation via graph coloring. However, the register allocator combines features from these two approaches with new features into a unique algorithm that does not resemble any of the other two allocators. In particular, a new algorithm to split live ranges was invented that applies the priority-based approach to the traditional splitting algorithm. The code conversion itself is quite simple once the results from the data-flow analysis and the register allocation are available.

The fourth chapter surveyed several parallel benchmark suites in order to determine a benchmark suite that is well suited for the evaluation of emulated multithreading. The survey used five different criteria to determine the suitability of a given benchmarks suite: The individual benchmarks should cover a wide range of computational problems, be widely used and possess well-known characteristics, scale to at least 64 processors, be available in source code, and have modest time and space requirements. Based on these criteria, the SPLASH2 benchmark suite was chosen for the evaluation of emulated multithreading on two different platforms: the Compaq XP1000 workstation and the Cray T3E massively parallel processor. From the four kernels and eight applications in this suite, three kernels and three applications were selected and subsequently ported to the Cray T3E. The algorithm, implementation, as well as the porting process for the six benchmarks was described in detail.

The fifth chapter describes the evaluation of multithreading on a single-processor system, i.e. the Compaq XP1000 workstation. After a detailed dis-

cussion of the hardware and software environments as well as the experimental methodology, the impact of the code conversion process on the size and structure of the original sources was examined. This investigation was based on the statistics for the original and converted assembler sources provided by the assembler converter. For all six benchmarks, the converted sources using super block optimization used the least number of instructions, although the converted parts of the code are still up to twice as large than their original counterparts. The changes in the instruction mix were described and explained in detail, especially the impact of the super block optimization.

Seven different versions were compared for every benchmark: base, g004, g016, g064, bblk, sblk, and posix. The latter six versions support multithreading and were executed with 1,2,4,8,16 threads in order to determine the impact of the number of threads on performance. In addition, each experiment was performed on three different problem sizes, i.e. the default problem size as well as twice and quadruple that size. Recall that there are no references to remote memory on this platform, i.e. no benefit to emulated multithreading. However, the results of the individual experiments are encouraging: The sblk version of all benchmarks is faster than the posix version in most cases, although the Tru64 operating system contains an efficient implementation of POSIX threads. In addition, the sblk version is only slightly slower than the base version with the exception of the fmm benchmark. These results indicate that emulated multithreading could be used to replace POSIX threads on single-processor or multi-processor machines with a shared-memory. For example, the current version of the widely used Apache http server uses POSIX threads and is an interesting candidate for the conversion to emulated multithreading.

The sixth chapter covers the evaluation of emulated multithreading on massively parallel processors, i.e. the Cray T3E. The corresponding experiments were performed on a Cray T3E-1200 installed at the Forschungszentrum Jülich. After a detailed description of the hardware and software environments as well as the experimental methodology, the results of the individual experiments were presented. For all six benchmarks, the same set of experiments that was used during the evaluation of emulated multithreading on the Compaq XP1000 were repeated on a single processing element of the Cray T3E. However, only the base, bblk, and sblk versions were taken into account. In the parallel case, the base, bblk, sblk versions of five different benchmarks were compared on a wide range of processor and thread numbers: For each benchmarks, the three versions were executed on 1,2,4,8,16,32, and 64 processors, the bblk and sblk were executed with 1,2,4,8, and 16 threads in each case. In addition, all experiments were performed on three different problem sizes, i.e. the default problem size as well as double and quadruple that size. The results for the fmm benchmark were omitted since all versions of the benchmarks were instable due to race conditions as described in Section 4.2.6.

The results of the individual experiments are quite encouraging: With the exception of the ocean benchmarks, the sblk version is always faster than the base version as long as no more than four threads are used. In some cases, the sblk version was significantly faster than the base version. The problems with the ocean benchmark might be caused by errors in the porting process, as even the base version scales poorly for larger number of processors and previous performance problems could be traced to an obscure bug in the assembler.

The results presented in the sixth chapter show that emulated multithreading is able to improve the performance of parallel applications by tolerating the latency of remote memory references. However, the current implementation does not even exploit the full potential of emulated multithreading. There are several ways to further increase the performance: For example, the main loop in the thread execution routine that is traversed once for each context switch, uses neither hand-coded optimizations nor prefetching. The communication routines in the emulation library are written in straightforward C code and should be ported to assembler code and optimized for the 21164 processor. The following paragraphs describe other techniques intended to decrease the overhead associated with emulated multithreading, thereby increasing performance.

Register partitioning divides the register set into multiple partitions and confines each thread to its own partition, thereby reducing the number of save and restore operations: As long as none of the threads executes an external call, the individual threads will not destroy the contents of other partitions, hence it is no longer necessary to save and restore registers from the thread descriptor. However, the number of spill operations might increase due to the smaller number of available registers in a partition, i.e. the increased register pressure. Register partitioning requires changes to the high-level language and assembler converters as well as the thread execution routine. These changes have already been implemented in the case of the assembler converter.

The results of the evaluation on single-processor and massively parallel processor machines reveal that super block optimization is important to the performance of emulated multithreading. The current algorithm used to create the super blocks is quite simple and can be improved by using profiling information to guide the creation of super blocks: Frequently traversed edges in the control-flow graph should not cross super block boundaries, otherwise a context switch has to be performed each time. Techniques known from trace scheduling could be used to implement the use of profiling information in an improved version of the algorithm. Instead of profiling information, static predication could also be used. Another way to increase the size of super blocks is to balance the ability to tolerate latency against the overhead caused by frequent context switches. This can be accomplished by checking for each remote memory reference whether the corresponding latency should be tolerated, i.e. whether the reference should force the end of a super block.

The current implementation of emulated multithreading would also benefit from tighter integration with a compiler. For example, this approach provides access to the profiling information used to create super blocks as well as the ability to steer the compiler to produce code that is well suited for emulated multithreading. In the case of register allocation and code scheduling, this kind of integration was shown to be useful [BEH91]. The advantages of integrated support for emulated multithreading in a compiler were already outlined in Section 3.8 using the SUIF compiler as an example. Apart from better opportunities for optimizations, the assembler and high-level language converter could reuse the existing compiler infrastructure. The next implementation of emulated multithreading should therefore be integrated into a compiler, preferably the SUIF compiler system.

Before working on the next implementation of emulated multithreading, the current implementation should be used in further evaluations in order to get a more complete picture of the performance characteristics: The number of benchmarks used during the evaluation of emulated multithreading should be increased in order to ensure that the results from Chapters 5 and 6 apply to a wide range of computational problems. The remaining six benchmarks of the SPLASH2 benchmark suite would be suitable candidates for such an evaluation, as these benchmarks could be ported quite easily based on the current experience. In addition, the impact of compiler optimizations on emulated multithreading should be investigated as some of the traditional optimizations were shown to be counter-productive for multithreaded applications [LEL+99]

Finally, emulated multithreading should be ported to other platforms since the Alpha architecture was discontinued recently. The most promising candidates would be the Power and Sparc architectures, since the MIPS and HP-PA architectures have been discontinued some time ago and the IA32 and IA64 architectures are not suitable for emulated multithreading. As parallel machines based on the Sparc architecture currently do not support more than 64 processors, the Power should be used as large-scale massively parallel processors based on this architecture are available. In the meantime, emulated multithreading could be ported to the AlphaServerSC [Cor00], a cluster of workstations or servers based on the Alpha architecture and the Tru64 operating system. This system combines the advantages of the two platforms used in the current evaluation of emulated multithreading: i.e. the 21264 processor with out-of-order execution and large first-level caches, and a massively parallel processor with a fast network that supports asynchronous communication. As the AlphaServer SC provides a software environment that is very similar to the existing platforms, the port should be possible in a small amount of time. Finally, emulated multithreading should be extended to cover MPI one-sided messages instead of the shared memory primitives from the shmem library as MPI is in much more widespread use than the proprietary shmem library, which is only available on Cray/SGI systems.

A. Alpha Architecture & Implementations

Using the formulation from Amdahl, Blauuw, and Brooks [ABBJ64], the Alpha Architecture Reference Manual [Com98] defines the terms *architecture* and *implementation* as follows:

Architecture is defined as the attributes of a computer seen by the machine-language programmer. This definition includes the instruction set, instruction formats, operation codes, addressing modes. and all register and memory locations that may be directly manipulated by a machine-language programmer.

Implementation is defined as the actual hardware structure, logic design, and data-path organization of the computer.

This chapter gives an overview of the Alpha architecture as well as the implementations of this architecture. Section A.1 describes the background of the Alpha architecture, while the architecture itself is described in Section A.2. The characteristics of past, present, and future implementations are specified in Section A.3. Systems based on these implementations are outside the scope of this chapter, information about theses systems can be found in various issues of the Digital or Compaq Technical Journals.

A.1 Introduction

Some background information is required in order to understand the individual characteristics of the Alpha architecture: The Alpha architecture was developed by Digital in order to provide an upgrade path to the existing VAX customer base. Section A.1.1 describes this successful CISC (Complex Instruction Set Computer) architecture as well as the problems associated with high-performance implementations. These problems motivated work on several RISC (Reduced Instruction Set Computer) projects within Digital. The various efforts were later merged into a single project (PRISM), which is sketched in Section A.1.2. After this project was canceled, work started on the Alpha architecture. The Section A.1.3 specifies the various design goals for the Alpha architecture project, while the Section A.2 introduces the architecture itself.

A.1.1 VAX Architecture

The VAX 11/780 was introduced in 1977 and was the first implementation of the VAX (Virtual Address eXtension) architecture. A detailed description of the architecture as well as the first implementation was published in [Str78]. The VAX architecture was designed to provide a 32 bit extension to the successful PDP-11 architecture, hence the name. The PDP-11 was a 16 bit system with a 64 Kbyte virtual address space, which started to be a limiting factor.

Compatibility with the older system was an important design goal for the VAX architecture: Apart from equivalent and identical instructions, data types, compilers, file systems and I/O busses, the VAX architecture provided a PDP-11 compatibility mode. During execution in this mode, the PDP-11 registers were mapped to native VAX registers and all addresses were sign-extended to 32 bit. This mode supported the whole PDP-11 instruction set with the exception of floating-point and privileged instructions. The most important difference between the two systems was the 4 Gbyte virtual address space provided by the VAX architecture.

The VAX architecture provides sixteen 32 bit general-purpose registers used by all instructions, i.e. there is no dedicated floating-point register file. Four registers have a special meaning: program counter, stack pointer, frame pointer, and argument pointer. The architecture uses four different access modes: kernel, executive, supervisor, and user. The user mode context contains the general-purpose registers and the processor status word, other resources are only accessible in one of the privileged modes. The processor status word stores the condition codes and is used to enable and disable the various traps.

The VAX architecture supports the usual integer and floating-point data types as well as bit fields, character and decimal strings. There are no restrictions to alignment, i.e. all data types may start on arbitrary byte locations, bit fields may start on arbitrary bit locations. The instruction set contains 304 instructions, later implementations support even more. The supported instructions include the usual integer, floating-point, and control instructions as well as queue, string, and context switch instructions. Almost every combination of instruction, data type, and addressing mode is possible, making the VAX instruction set highly orthogonal. This leads to a complex instruction format: one or two bytes of operation code followed by operand specifiers (2 to 10 bytes) for up to three operands. Instruction length therefore varies between 1 and 32 bytes, which complicates instruction decoding.

The VAX architecture provides a rich set of addressing modes: immediate, register direct with offset, post-increment or post-decrement, register indirect with offset or post-increment, as well as indexed addressing. Indexed addressing is only used in combination with one of the aforementioned addressing modes, the final operand address is determined by adding the contents of

the specified register (multiplied by operand size) to the base register of the original mode.

Although the VAX architecture provides a rich functionality, studies show that a lot of this functionality was infrequently used [Wie82][CL82].

A.1.2 Digital RISC Projects

The VAX architecture has a large number of intra- and inter-instruction dependencies, making high-performance implementations difficult: intra-instruction dependencies complicate pipelining, whereas inter-instruction dependencies make super-scalar and out-of-order implementations almost impossible. Similar restrictions exist in other CISC architectures. During the early 80s, RISC (Reduced Instruction Set Computer) as proposed by Patterson [PD80] promised to overcome these restrictions by using a fundamentally different approach.

The first Digital RISC project started in 1982 to explore the feasibility of an ECL (Emitter Coupled Logic) RISC implementation. A comparison between this implementation and a VAX architecture implementation based on similar technology proved favorable to the RISC approach [BC91b]. Several RISC projects were subsequently started within Digital: SAFE (Streamlined Architecture for Fast Execution), HR-32 (Hudson RISC) and CASCADE. These efforts were later combined into the PRISM (Parallel RISC Machine) project which is described in the next paragraph.

The PRISM project defined a 32 bit as well as a 64 bit architecture. Both architectures provide 64 general-purpose registers for integer and floating-point instructions, i.e. there are no split register files. All operations are performed between these registers, memory is referenced by load and store instructions only. Instructions have a fixed length and use up to three operands. There are no condition codes, branch instructions operate on registers instead. Data types are limited to VAX longword (32 bit version), VAX quadword (64 bit version), VAX F and G floating-point formats, i.e. there is no support for byte and word data types. An interesting feature is the support for vector arithmetic with 16 vector registers, each 64 elements wide, and corresponding vector instructions. In summary, the PRISM architecture is a typical load/store RISC architecture with the exception of the combined register file and support for vector arithmetic. A detailed description of the PRISM architecture was published in [BOW+90].

The PRISM project developed an implementation of the 32 bit PRISM architecture intended for Digital's workstation line. Unfortunately, the MIPS architecture was chosen instead, hence this implementation was never commercialized. A brief description of the implementation was published in [CDD+89], the important characteristics are summarized here: The implementation uses a 5-stage pipeline with fixed instruction latencies and in-order instruction issue, but out-of-order completion. The chip was fabricated in a

1.5 µm CMOS process, using approximately 300 000 transistors and a maximum clock frequency of 50 MHz.

A.1.3 Design Goals

One of the first publications regarding the Alpha architecture [Sit92] lists four design goals for the architecture:

- High Performance
- Longevity
- Capability to run both VMS and UNIX operating systems
- Easy migration from VAX and MIPS architectures

Recent publications, e.g. the Alpha Architecture Reference Manual [Com98], list slightly different goals:

- Longevity
- High Performance
- Adaptability
- Scalability

Considering the Alpha architecture's background, the first version seems more accurate, whereas the second versions seems to reflect the current goals for the Alpha architecture.

High Performance and Longevity. The Alpha architects set a 15- to 25-year design horizon for the new architecture. Based on their experience with the PDP-11 and VAX architectures, especially the lack of address space, the Alpha architecture was designed as a full 64 bit Architecture from the start. Furthermore, the architects tried to avoid anything that might limit future performance: The instruction set is subsettable, i.e. support for specific instruction subsets can be added to or removed from future versions of the architecture. For example, support for VAX floating-point data types is one of these subsets that may be removed in the future.

Instruction encodings that are not occupied are implemented as traps or null operations in current implementations, such that code written for future implementations that support additional instructions can still be executed on todays processors via software emulation. In addition, the internal fields of the architecture were generously sized for later expansion.

Since computer performance had increased by a factor of 1000 in the past 25 years, the Alpha architecture was designed to allow a similar performance improvement during its life-time. This improvement was to come from three different sources:

Clock frequency was to provide a factor of ten improvement in performance during the architecture's life-time. At the time of writing, implementations of the Alpha architecture have already achieved a factor of 7

improvement in clock frequency: from 150 MHz for the first implementation (21064) to 1 GHz for the latest implementation (21264B). According to the current road-maps, the goal of ten-fold improvement in clock frequency will be achieved in the 2001/2002 time-frame, i.e. significantly before the projected life-time. These performance improvements are not caused by a degenerate baseline, i.e. an initial implementation using a slow clock. Rather, implementations of the Alpha architecture have been using the fastest clock frequencies among all microprocessors until recently.

Super-scalar execution was to provide another factor of ten improvement by executing multiple instructions in a single clock cycle. In order to enable this level of super-scalar execution, the Alpha architecture avoids special processor state like branch condition codes and special registers. The only special resources in the Alpha architecture are the dedicated state required by the IEEE 754 floating-point standard [IEE85] for dynamic rounding of floating-point values as well as the multiprocessing primitives as described below. In particular, the Alpha architecture has no branch delay slots like other RISC architectures, no byte and word memory references, and no precise exceptions. However, byte and word memory references as well as precise exceptions were later added to the architecture.

Since the first implementation (21064A) issued up to two instructions per cycle and current implementations issue up to six (four sustained) instructions per cycle, there is only a factor of three improvement up to now. It has proved difficult to extract the available amount of instruction level parallelism, hence these small improvements. Future implementations of the Alpha architecture, e.g. the 21464, were planned to employ simultaneous multithreading to increase the number of instructions that can be issued in parallel.

Parallel processing was to provide the remaining factor of ten improvement in performance. Therefore the Alpha architecture was designed with multiprocessing capabilities from the outset. The architecture provides load-locked and store-conditional primitives to support atomic reads and writes to shared memory. The underlying memory model uses relaxed consistency with explicit barrier instructions to enforce strict ordering between memory references. Systems based on the Alpha architecture use up to 32 processors in shared memory systems like the AlphaServer GS320 [GSSD00], and up to 2048/4096 processors in distributed memory machines like the Cray T3E [Oed96] and Compaq AlphaServer SC [Cor00]. The next implementation of the Alpha architecture, i.e. the 21364, will provide integrated support for 64-way shared memory multiprocessing. However, the performance improvement in practice is hard to determine since parallel speedup is largely application-specific.

Operating systems and Migration. One of the major goals in developing the Alpha architecture was to provide the existing MIPS/OSF and VAX/VMS customer base with an upgrade path to the new architecture. The Alpha architecture uses PALcode to decouple the architecture from operating system-specific issues. PALcode is a set of low-level functions, e.g. page table refill and context switch, that are tailored to the operating system. These routines are executed in a special environment with disabled interrupts, no memory translation, and access to additional state that is provided by the implementation. The PALcode routines reside in main memory and can therefore be updated at runtime.

Another feature for operating system migration is the support for VAX integer and floating-point data formats. The corresponding instructions are subsettable, i.e. these instructions may be removed in future versions of the architecture. Binary translation was used to execute MIPS/OSF and VAX/VMS binaries on the Alpha architecture under the OSF and VMS operating systems, respectively. Detailed descriptions of the translation process were published at the launch of the Alpha architecture [SKMR92].

Adaptability and Scalability. As mentioned above, instead of independence from operating systems and ease of migration, later revisions of the Alpha architecture manual state adaptability and scalability as goals for the Alpha architecture. It is likely that the earlier goals are no longer mentioned since the migration from the VAX and MIPS architectures has been largely finished by now. Adaptability means that the Alpha architecture is largely independent from operating system and implementation details. Therefore PALcode remains an important feature of the architecture. Scalability means that the Alpha architecture can be used in low-cost systems as well as high-performance systems. Early implementations of the Alpha architecture were used in such diverse areas as embedded computing and massively parallel computers, but later implementations have focused on high-performance systems.

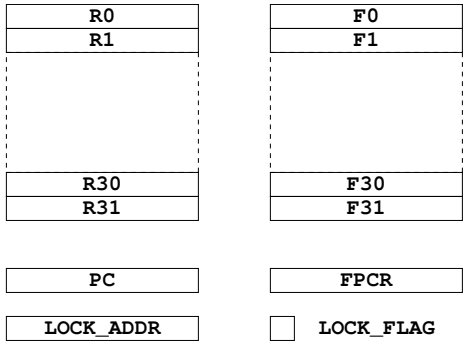
A.2 Alpha Architecture

This section describes the Alpha architecture in detail. Section A.2.1 covers the architecture state as seen by the machine-language programmer, while Section A.2.2 covers the supported address and data formats. The syntax and semantics of the individual instructions are described in Section A.2.3. Sections A.2.4 provides information about the PALcode environment.

A.2.1 Architecture State

The architecture state as seen by the machine-language programmer is depicted in Figure A.1. There are 32 integer registers (R0-R31) and 32 floating-point registers (F0-F31), each 64 bits wide. Registers R31 and F31 are zero

Fig. A.1. Alpha Architecture State



source and sink registers, i.e. reads to these registers return zero and values written to these registers are discarded. The program counter (PC) contains the virtual address of the current instruction. The program counter is incremented by every instruction, branch-type instructions can modify the program counter in other ways.

The floating point control register (FPCR) is 64 bits wide, most of these bits are reserved for future use. The remaining bits contain status and control information for the different floating-point trapping modes as well as the rounding mode for instructions using dynamic rounding mode. Apart from the IEEE $+\infty$ rounding mode, all rounding modes can be specified explicitly in the instruction, therefore the floating-point control register is seldom used during normal operation.

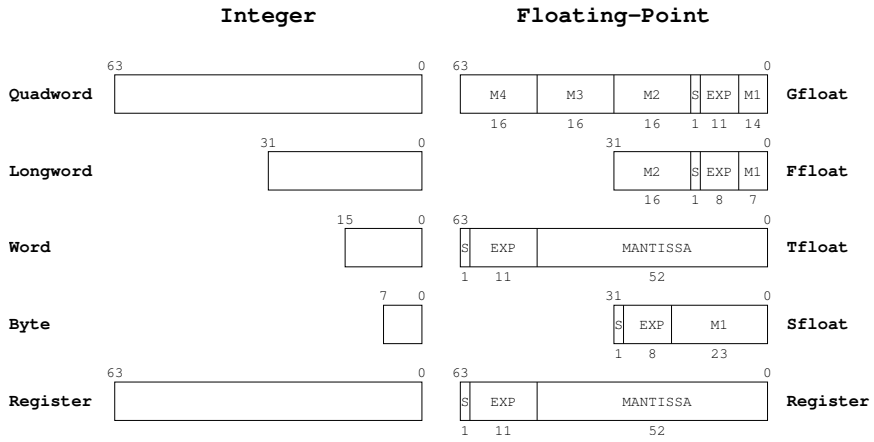
Additional state is available in the PALcode environment, the amount of state depends on the implementation. Typical implementations provide additional registers, processor status words, stack pointers, address and data translation buffers, as well as other implementation-specific state.

A.2.2 Address, Data and Instruction Formats

Address Format. The Alpha architecture uses the byte, i.e. 8 bits, as the basic addressable unit. Virtual addresses are 64 bits wide, forming a virtual address space of 16 Exabytes. Implementations may restrict the effective size of virtual addresses by using identical high-order bits, but at least 43 bits must be distinguishable, yielding a maximum virtual address space of 8 Terabyte. Note that all implementations have to check all 64 bits for virtual addresses, independent of the size of the implemented virtual address space.

Early implementations support only the minimum virtual address space, while current implementations distinguish up to 48 bits, yielding a maximum virtual address space of 256 Terabyte. The Alpha architecture supports multiple disjoint address spacing by using additional address space numbers

Fig. A.2. Alpha Architecture Data Formats



(ASN). The size of the physical address space depends on the implementation and is usually a lot smaller: Early implementations support 34 bits while current implementations support up to 44 bits.

The default byte ordering is little endian, i.e. the individual bytes within a quadword are numbered from right to left. Big endian support, i.e. numbering the bytes from left to right, is optional in the Alpha architecture. However, all implementations support this option so far. Note that the instruction stream is always accessed in little endian order regardless of the chosen endianness.

Data Formats. The most important data types supported by the Alpha architecture are the quadword, longword, word, and byte integers as well as the VAX Ffloat, Gfloat, and the IEEE Sfloat, Tfloat floating-point data types. The memory and register layout of these data types is depicted in Figure A.2. Note that the register format is the same for all data types within the same category. The following paragraphs describe the individual data types in more detail.

A byte contains eight contiguous bits starting on an arbitrary byte boundary. This data type is only supported by a few instructions, namely load and store, sign extend, and byte manipulation instructions. A word contains two contiguous bytes starting on an arbitrary byte boundary. The address of a word equals the address of the byte containing the least significant bit. Like the byte, the word is only supported by load and store, sign-extend, and byte manipulation instructions. The byte and word data types were not supported in early revisions of the Alpha architecture, this support was added later with the byte and word extension (BWX).

A longword contains four contiguous bytes starting on an arbitrary byte boundary. The address of a longword equals the address of the byte containing the least significant bit. Longwords are always treated as two's-complement

integers, i.e. there are no unsigned longwords. The quadword contains eight contiguous bytes starting on an arbitrary byte boundary. The address of a quadword equals the address of the byte that contains the least significant bit. The quadword represents the fundamental data type in the Alpha architecture and is therefore supported by all types of instructions.

The VAX Ffloat and Gfloat floating-point format contain four and eight contiguous bytes starting on an arbitrary byte boundary, respectively. The address of these data types equals the address of the byte containing the least significant bit. The Ffloat floating-point data type consists of a sign bit s, an eight bit exponent e represented in base-128 format as well as 24 bits of normalized mantissa m. The Gfloat floating-point data type consists of a sign bit s, an eleven bit exponent e represented in base-1024 format as well as 53 bits of normalized mantissa m. In both cases, the most significant bit is only implied, but not stored in memory. The values v_1 represented by a Ffloat and the value v_2 represented by a Gfloat floating-point data type are determined by the following formulas:

$$v_1 = (-1)^s \cdot 2^{e-128} \cdot \left(1 + \sum_{i=0}^{23} m_{23-i} \cdot 2^{-i}\right)$$
$$v_2 = (-1)^s \cdot 2^{e-1024} \cdot \left(1 + \sum_{i=0}^{52} m_{52-i} \cdot 2^{-i}\right)$$

Note that there are two special cases: A zero sign bit together with a zero exponent represent the value 0, while a sign bit of one together with a zero exponent is taken as a reserved operand, causing an arithmetic exception if used by a floating-point instruction.

The IEEE Sfloat and Tffloat floating-point format contain four and eight contiguous bytes starting on an arbitrary byte boundary, respectively. The address of these data types equals the address of the byte containing the least significant bit. The IEEE Sfloat data type consists of a sign-bit s, an eight bit exponent e represented in base-127 format as well as 24 bits of normalized mantissa m. The most significant bit is only implied, but not stored in memory. The IEEE Tfloat data type consists of a sign-bit s, an eleven bit exponent e represented in base-1023 format as well as 52 bits of normalized mantissa m. The values v_1 represented by a Sfloat and the value v_2 represented by a Tfloat floating-point data type are determined by the following formulas:

$$v_1 = (-1)^s \cdot 2^{e-127} \cdot \left(1 + \sum_{i=0}^{23} m_{23-i} \cdot 2^{-i}\right)$$
$$v_2 = (-1)^s \cdot 2^{e-1023} \cdot \sum_{i=0}^{52} m_{52-i} \cdot 2^{-i}$$

Note that there are several special cases: A zero exponent together with a zero mantissa represent the values +0 or -0, depending on the sign bit. In the case of an zero exponent and a non-zero mantissa, the represented values are determined by the following formulas:

$$v_1 = (-1)^s \cdot 2^{-126} \cdot \sum_{i=0}^{23} m_{23-i} \cdot 2^{-i}$$

$$v_2 = (-1)^s \cdot 2^{-1022} \cdot \sum_{i=0}^{52} m_{52-i} \cdot 2^{-i}$$

An exponent of all ones together with a zero mantissa represents $+\infty$ or $-\infty$, depending on the sign bit. An exponent of all ones together with a non-zero mantissa represents a Not-A-Number (NaN). NaNs come in two different forms: Quiet NaNs and Signaling NaNs. The former propagates through almost all operations without generating an arithmetic exception, while the latter signals an invalid operation when used by an arithmetic instruction and may cause an arithmetic exception.

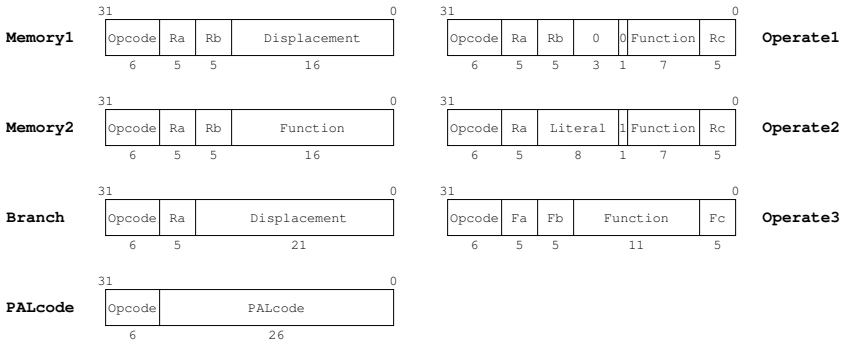
Apart from the data types mentioned above, the Alpha architecture has limited support for the VAX Dfloat floating-point data type, i.e. load, store, and conversion to Gfloat. The three least significant bits of the mantissa are lost during the conversion process. Support for this data type enables the processing of data files produced by legacy applications under VAX/VMS. The IEEE Xfloat floating-point data type is currently only supported in software, but the Alpha architecture defines the necessary register and memory formats. This ensures consistency with possible future implementations that may support this data type.

Instruction Formats. The Alpha Architecture uses seven different instruction formats: two memory instruction formats, a branch instruction format, three operate instruction formats, and a PALcode instruction format. All instruction formats are 32 bit wide with a common opcode field located in the most significant six bits of the instruction. The individual instruction formats are depicted in Figure A.3 and are described in the following paragraphs.

The first memory instruction format is used for load and store instructions and indirect branches. Apart from the common 6 bit opcode, the format contains two 5 bit register addresses and a 16 bit displacement. The displacement is used by load and store instructions to specify the effective address: The displacement is sign-extended and added to the contents of the source register Rb, ignoring any overflows. Indirect branches use the displacement to provide hints to branch prediction logic, the meaning depends on the individual instruction and is explained in Section A.2.3.

The second memory instruction format is used for a set of miscellaneous instructions. It differs from the first memory instruction format by substi-

Fig. A.3. Alpha Architecture Instruction Formats



tuting the displacement with a function code of the same size. The common opcode together with this function code specifies the individual instruction. Some instructions may use less than two register addresses, the unused fields must contain the address of register R31. This principle applies to all instructions defined by the Alpha architecture, floating-point instructions specify the address of register F31 instead.

The branch instruction format is used for PC-relative branches and subroutine calls. Apart from the common 6 bit opcode, it contains one 5 bit register address and a 21 bit displacement. The displacement is used to form the virtual address of the target branch or call location by adding the sign-extended displacement to the contents of the updated program counter. Note that the updated program counter is equal to the address of the current instruction plus four, the size of a single instruction. The displacement is interpreted as an instruction offset since the target branch or call location must start on an instruction boundary, hence the displacement is multiplied by four prior to addition.

PALcode instructions use the PALcode instruction format. This format contains only the common opcode and a 26 bit function code.

The first operate format is used for integer instructions that utilize three register operands. Apart from the common 6 bit opcode, the format contains two 5 bit register addresses Ra and Rb for the source operands, a 5 bit register address Rc for the destination operand, and a 7 bit function code. Bit number 12 distinguishes between this format and the second operate format described below. The three unused bits are zero for all instructions, at least in the current revision of the Alpha architecture.

The second operate instruction format is similar to the first one and is used for integer instructions that use two register operands and one literal. This format substitutes the source register address Rb and the three unused bits above with an eight bit literal that is zero-extended before use.

The third operate instruction format is used for instructions that operate on the floating-point register set. Apart from the common 6 bit opcode, it contains two 5 bit source register addresses, one 5 bit destination register address, and an 11 bit function code. This function code together with the common opcode selects the individual instruction.

A.2.3 Instruction Set

The instructions defined by the Alpha architecture can be grouped into several disjoint sets: integer memory, control, arithmetic, logical and shift, as well as floating-point memory, control, and arithmetic instructions. Later revisions of the Alpha architecture define additional instructions in the following four extensions: byte and word extension (BWX), multimedia extension (MVI), floating-point extension (FIX), and count extension (CIX). The following sections describe the individual instructions in the basic sets as well as the current extensions.

Integer Memory Instructions. The integer memory instructions transfer data between memory and integer registers, with the exception of the LDA/LDAH instruction pair. The individual instructions in this set are listed in Table A.1. All instructions in this set do not interpret the transferred data in any way, hence no arithmetic exceptions are possible.

The LDA/LDAH instruction pair is used to form constant values: The LDA instruction places the 16 bit displacement in the destination register, while the LDAH instruction performs a 16 bit left shift first. Almost every 32 bit constant can be constructed using this instruction pair, three instructions are required for some constants. Larger constants should be loaded from memory.

The load and store instructions support longwords (LDL/STL) and quadwords (LDQ/STQ) and come in three different forms: The normal LDx/STx instructions transfer data from and to memory at the virtual address given by adding source register and displacement to and from the destination register. The unaligned LDx_U/STx_U form ignores the three least-significant bits of the virtual address, i.e. accesses the aligned longword and quadword that contains the desired locations. These instructions are used to replace accesses to unaligned memory locations by a short sequence of instructions.

The LDx_L/STx_C instruction pairs enable atomic updates of longwords and quadwords: If the LDx_L instruction succeeds, it stores the translated virtual address in the per-processors lock address register described in Section A.2.1 and sets the per-processor lock flag. A subsequent STx_C instruction only performs the store operation if the lock flag is still set and the lock address register points to the same address. The outcome of the store instruction is recorded in the source register, the per-processor lock flag is cleared in all cases. Using these instructions, an atomic read-modify-write access can be implemented as follows:

Table A.1. Integer Memory Instructions

Name	Description
LDA	Load Address
LDAH	Load Address High
LDL	Load Sign-Extended Longword
LDL_L	Load Sign-Extended Longword Locked
LDQ	Load Quadword
LDQ_L	Load Quadword Locked
LDQ_U	Load Quadword Unaligned
STL	Store Longword
STL_C	Store Longword Conditional
STQ	Store Quadword
STQ_C	Store Quadword Conditional
STQ_U	Store Quadword Unaligned

```
L1:
   LDQ_L   R1, disp(R2)
   (modify R1)
   STQ_C   R1, disp(R2)
   BEQ     R1, L1
```

To ensure that the above sequence eventually succeeds, several programming considerations have to be met: The LDx_L/STx_C instruction pair must access the same address region and the modify sequence must not generate any exceptions. The size of address range depends on the implementation and ranges from 16 bytes to the size of a physical page. Exception handlers clear the per-processor lock flag, hence the modify sequence should not contain any other memory operations. In addition, the modify sequence should contain less than 40 instructions, otherwise the sequence may always fail due to timer interrupts.

Integer Control Instructions. Three different forms of integer control instructions are available: conditional branches, unconditional branches, and jumps. The individual instructions in this set are listed in Table A.2 and are described in the following paragraphs.

Conditional branches test the source register and add the displacement to the contents of the updated program counter if the specified relationship is true, otherwise the program counter remains unchanged. The following relationships are supported: $=$ (BEQ), $\geq$ (BGE), $>$ (BGT), $\leq$ (BLE), $<$ (BLT), $\neq$ (BNE), as well as even (BLBC) and odd (BLBS).

The unconditional branches store the contents of the current program counter in the specified destination register and update the program counter like their conditional counterparts. The BR and BSR instructions perform identical operations, but differ in their hints to the branch prediction logic: In contrast to BR, BSR pushes the contents of the updated PC to the return-address stack.

Table A.2. Integer Control Instructions

Name	Description
BEQ	Branch Equal to Zero
BGE	Branch Greater Than or Equal to Zero
BGT	Branch Greater Than Zero
BLBC	Branch Low Bit Clear
BLBS	Branch Low Bit Set
BLE	Branch Less Than or Equal to Zero
BLT	Branch Less Than Zero
BNE	Branch Not Equal to Zero
BR	Branch
BSR	Branch to Subroutine
JMP	Jump
JSR	Jump to Subroutine
RET	Return from Subroutine
JSR_COROUTINE	Jump to Subroutine Return

The indirect branch instructions store the contents of the updated program counter in the specified destination register, while the program counter is loaded from the specified source register. Similar to the unconditional branches above, all indirect branch instructions perform identical operations, they differ only in the hints to branch-prediction logic: The JMP and JSR instructions use the 14 least-significant bits of the displacement to form a PC-relative hint that can be used to prefetch the instruction stream at the target location. In addition, the JSR instruction pushes the contents of the updated program counter to the return-address stack. The RET and JSR_COROUTINE instructions pop the return-address stack and use this value as the predicted target location. In addition, the JSR_COROUTINE instruction pushes the contents of the updated program counter to the return-address stack.

Integer Arithmetic Instructions. Integer arithmetic instructions include the following operations: add, subtract, multiply, as well as signed and unsigned comparisons. The individual instructions are listed in Table A.3 and are described in the following paragraphs.

The ADDx and SUBx instructions perform integer addition and subtraction and are available in three different forms: normal, scaled-by-4 and scaled-b-8. The latter two operations multiply the second source operand by 4 or 8 without checking for overflows, respectively. The CMPxx compare instructions can be signed and unsigned and support the $=$ (CMPEQ), $\leq$ (CMPLE, CMPULE) and $<$ (CMPLT, CMPULT) relations. Note that the CMPEQ instruction can be used for signed and unsigned comparisons and that the missing $\geq$, $>$ relations are supported by reversing the order of the source operands. The MULx instruction performs integer multiplication. Since the product of two 64 bit integers can be up to 128 bit large, the MULx instruction may cause an arithmetic exception if an overflow occurs, i.e. the result

Table A.3. Integer Arithmetic Instructions

Name	Description
ADDL	Add Longwords
ADDQ	Add Quadwords
S4ADDL	Add Longwords, Scaled by 4
S4ADDQ	Add Quadwords, Scaled by 4
S8ADDL	Add Longwords, Scaled by 8
S8ADDQ	Add Quadwords, Scaled by 8
CMPEQ	Integer Compare, Equal to Zero
CMPLE	Integer Compare, Less Than or Equal to Zero
CMPLT	Integer Compare, Less Than Zero
CMPULE	Integer Compare, Unsigned Less Than or Equal to Zero
CMPULT	Integer Compare, Unsigned Less Than Zero
MULL	Multiply Longwords
MULQ	Multiply Quadwords
UMULH	Multiply Quadwords, Unsigned High
SUBL	Subtract Longwords
SUBQ	Subtract Quadwords
S4SUBL	Subtract Longwords, Scaled by 4
S4SUBQ	Subtract Quadwords, Scaled by 4
S8SUBL	Subtract Longwords, Scaled by 8
S8SUBQ	Subtract Quadwords, Scaled by 8

is larger than $2^{63} - 1$. In this case the UMULH instruction can be used to generate the upper 64 bits of the product.

Integer Logical and Shift Instructions. The set of integer logical and shift instructions contains boolean operations, conditional moves, and shift instructions. The individual instructions are listed in Table A.4 and are described in the following paragraphs.

The following boolean operations are supported: AND, OR, and XOR. In addition, each of the three operations is supported with the complement of the second operand, i.e. BIC, ORNOT, EQV. Note that the missing NOT operation can be substituted by ORNOT with a zero source operand. The conditional moves test the first source register for the specified relation: if the relationship is true, the contents of the second source operand are moved to the destination register. The conditional moves support the same set of relations like the conditional branches: $=$ (CMOVEQ), $\geq$ (CMOVGT), $>$ (CMOVGE), $\leq$ (CMOVLE), $<$ (CMOVLT), $\neq$ (CMOVNE), as well as even (CMOVLBC)) and odd (CMOVLBS). The shift instructions are available in two different forms: arithmetic and logical. The logical shifts use zero to fill the vacated bit positions, while the arithmetic right shift uses the most significant bit. There is no arithmetic left shift, since a logical left shift is sufficient as long as no overflow occurs. In that case an integer multiply with 2^s should be used instead, where s is the shift distance.

Floating-Point Memory Instructions. The floating-point memory instructions transfer data between memory and floating-point registers. The

Table A.4. Integer Logical & Shift Instructions

Name	Description
AND	Logical Product
BIC	Logical Product with Complement
BIS	Logical Sum
EQV	Logical Equivalence
ORNOT	Logical Sum with Complement
XOR	Logical Difference
CMOVEQ	Integer Conditional Move, Equal to Zero
CMOVGE	Integer Conditional Move, Greater Than or Equal to Zero
CMOVGT	Integer Conditional Move, Greater Than Zero
CMOVLBC	Integer Conditional Move, Low Bit Clear
CMOVLBS	Integer Conditional Move, Low Bit Set
CMOVLE	Integer Conditional Move, Less Than or Equal to Zero
CMOVLT	Integer Conditional Move, Less Than Zero
CMOVNE	Integer Conditional Move, Not Equal to Zero
SLL	Logical Left Shift
SRA	Arithmetic Right Shift
SRL	Logical Right Shift

individual instructions in this set are listed in Table A.5. Note that all instructions in this set do not interpret the transferred data in any way, hence no arithmetic exceptions can occur. The load and store instructions support Ffloats (LDF/STF), Gfloats (LDG/STG), Sfloats (LDS/STS) and Tfloats (LDT/STT). All LDx/STx instructions behave similar to their integer counterparts and transfer data from and to memory at the virtual address given by adding source register and displacement to and from the destination register. The LDG/STG instruction pair should be used to load and store Dfloat values, since the required reordering of bits is identical for both data types.

Floating-Point Control Instructions. There are neither unconditional nor indirect floating-point branch instructions, since unconditional branches and jumps do not test any source operands. The individual instructions in this set are listed in Table A.6 and operate as follows: The FBxx instructions test the source operand for the specified relationship and perform a PC-relative branch like their integer counterparts if this relationship is true. The floating-point conditional branches support the same set of relations with the exception of the odd and even relations, which make no sense for floating-point operands. Note that the test is only based on the sign bit and whether the rest of the operand is all zero, i.e. the data is not interpreted in any way.

Floating-Point Arithmetic Instructions. The floating-point arithmetic instructions contain floating-point arithmetic, comparison, and conditional move instructions, among others. The individual instructions are listed in Table A.7 and are described in the following paragraphs.

The CPYSx instructions copy the sign and/or exponent between source and destination operands. The MT_FPCR and MF_FPCR instructions provide access to the floating-point control register. Note that the latter two

Table A.5. Floating-Point Memory Instructions

Name	Description
LDF	Load Ffloat
LDG	Load Gfloat (Dfloat)
LDS	Load Sfloat (Longword)
LDT	Load Tfloat (Quadword)
STF	Store Ffloat
STG	Store Gfloat (Dfloat)
STS	Store Sfloat (Longword)
STT	Store Tfloat (Quadword)

Table A.6. Floating-Point Control Instructions

Name	Description
FBEQ	Floating-Point Branch, Equal to Zero
FBGE	Floating-Point Branch, Greater Than or Equal to Zero
FBGT	Floating-Point Branch, Greater Than Zero
FBLE	Floating-Point Branch, Less Than or Equal to Zero
FBLT	Floating-Point Branch, Less Than Zero
FBNE	Floating-Point Branch, Not Equal to Zero

instructions must be enclosed in exception barriers to guarantee proper access to the floating-point control register.

There are four different forms of the ADDx, DIVx, MULx, and SUBx arithmetic instructions, one for each floating-point data type. These instructions perform floating-point addition, division, multiplication, and subtraction, respectively. The rounding and trapping modes can be explicitly specified for every instruction with the exception of the IEEE $+\infty$ rounding mode, which is only available via the floating-point control register by using dynamic rounding mode.

The CMPGxx and CMPTxx instructions perform comparisons on Gfloat and Tfloat data types, respectively. These instructions support the same set of relations as the floating-point conditional branches. An exception is the CMPTUN instructions which can be used to check for NaNs and is only provided for the Tfloat data type, since the VAX floating-point data types do not support NaNs. Note that separate comparison instructions for the Ffloat and Sfloat data types are not necessary, since the register layout for all floating-point data types is the same.

There is a rich set of conversion instructions: The CVTLQ/CVTQL instructions perform integer conversion between longwords and quadwords for integers stored in floating-point registers, respectively. The CVTQx instructions convert quadwords into the Ffloat, Gfloat, Sfloat and Tfloat floating-point data types, the reverse operation is performed by the CVTTQ instruction for all IEEE and VAX floating-point data types. The CVTTS and CVTST instructions convert between the IEEE Sfloat and Tfloat data types, while the CVTGF instruction converts between the VAX Gfloat and Ffloat data

Table A.7. Floating-Point Arithmetic Instructions

Name	Description
CPYS	Copy Sign
CPYSE	Copy Sign & Exponent
CPYSN	Copy Sign Negate
CVTLQ	Convert Longword to Quadword
CVTQL	Convert Quadword to Longword
FCMOVEQ	FP Conditional Move, Equal to Zero
FCMOVGE	FP Conditional Move, Greater Than or Equal to Zero
FCMOVGT	FP Conditional Move, Greater Than Zero
FCMOVLE	FP Conditional Move, Less Than or Equal to Zero
FCMOVLT	FP Conditional Move, Less Than Zero
FCMOVNE	FP Conditional Move, Not Equal to Zero
MF_FPCR	Move from Floating-Point Control Register
MT_FPCR	Move to Floating-Point Control Register
ADDF	Add Ffloats
ADDG	Add Gfloats
ADDS	Add Sfloats
ADDT	Add Tfloats
CMPGEQ	Compare Gfloat, Equal to Zero
CMPGLE	Compare Gfloat, Less Than or Equal to Zero
CMPGLT	Compare Gfloat, Less Than Zero
CMPTEQ	Compare Tfloat, Equal to Zero
CMPTLE	Compare Tfloat, Less Than or Equal to Zero
CMPTLT	Compare Tfloat, Less Than Zero
CMPTUN	Compare Tfloat, Unordered
CVTDG	Convert Dfloat to Gfloat
CVTGD	Convert Gfloat to Dfloat
CVTGF	Convert Gfloat to Ffloat
CVTGQ	Convert Gfloat to Quadword
CVTQF	Convert Quadword to Ffloat
CVTQG	Convert Quadword to Gfloat
CVTQS	Convert Quadword to Sfloat
CVTQT	Convert Quadword to Tfloat
CVTST	Convert Sfloat to Tfloat
CVTTQ	Convert Tfloat to Quadword
CVTTS	Convert Tfloat to Sfloat
DIVF	Divide Ffloats
DIVG	Divide Gfloats
DIVS	Divide Sfloats
DIVT	Divide Tfloats
MULF	Multiply Ffloats
MULG	Multiply Gfloats
MULS	Multiply Sfloats
MULT	Multiply Tfloats
SUBF	Subtract Ffloats
SUBG	Subtract Gfloats
SUBS	Subtract Sfloats
SUBT	Subtract Tfloats

types. Note that a conversion between Ffloat and Gfloat data types is not necessary, since both formats use an identical register layout and a Ffloat always fit in a Gfloat. The CVTGD and CVTDG conversion instructions are used to convert the partially supported VAX Dfloat data type to and from the Gfloat data type. Together with the load and store instructions for this data type, data files produced by legacy applications under VAX/VMS can be processed.

The floating-point conditional moves FCMOVxx behave like their integer counterparts. These instructions support the same set of relations like the floating-point conditional branches. As already pointed out above, the test is only based on the sign bit and whether the rest of the operand is all zero, i.e. the data is not interpreted in any way.

Miscellaneous Instructions. The individual instructions that do not fall into any of the other categories are listed in Table A.8. The AMASK instruction is used to check for the presence of extensions to the Alpha architecture instruction set, namely the BWX, MVI, FIX and CIX extensions described in later sections. The IMPLVER instruction performs a similar task as it is used to determine the major implementation version of the executing processor. The IMPLVER instruction should be used to make code-tuning decisions, while the AMASK instruction should be used to make instruction set decisions.

The EXCB exception barrier does not issue until all integer and floating-point exceptions and updates to the floating-point control register have been completed. The EXCB is a superset of the TRAPB instruction, which does not issue until all prior instructions are guaranteed to complete without causing arithmetic traps.

The MB memory barrier is used to ensure the proper ordering of load and store operation in multiprocessor systems. The barrier guarantees that all prior load and store instructions have accessed memory, as observed by other processors, before subsequent load and store instructions access memory. The MB barrier is a superset of the WMB barrier, which restricts ordering to stores only. Note that the coherence of the instruction stream is managed by the IMB instruction memory barrier that is realized in PALcode.

The ECB, WH64, FETCH and FETCH_M instructions provide hints to the memory system about future access patterns: The ECB instruction hints that the specified virtual address will not be accessed in the near future and should therefore be moved to a lower part in the memory hierarchy to allow the reuse of cache resources. The FETCH instructions hint that the aligned 512 byte block around the specified virtual address will be accessed in the near future and should therefore be moved to a higher point in the memory hierarchy in order to reduce memory latency. In addition, the FETCH_M instruction hints that all or part of the 512 byte block will be modified. The WH64 instruction hints that the aligned 64 byte block surrounding the spec-

Table A.8. Miscellaneous Instructions

Name	Description
AMASK	Architecture Mask
CALL_PAL	Call PALcode Routine
ECB	Evict Cache Block
EXCB	Exception Barrier
FETCH	Prefetch Data
FETCH_M	Prefetch Data, Modify Intent
IMPLVER	Implementation Version
MB	Memory Barrier
RPCC	Read Processor Cycle Count
TRAPB	Trap Barrier
WH64	Write Hint
WMB	Write Memory Barrier

ified virtual address will never be read again, but will be overwritten in the near future.

The RPCC instruction returns the current value of the processor cycle counter that increments every clock cycle. Note that the counter is only 32 bits wide, therefore care must be taken to detect any wrap-arounds between two accesses to the cycle counter. The CALL_PAL instruction is used to execute PALcode functions as described in section A.2.4.

Byte and Word extension (BWX). The byte and word extension to the Alpha architecture instruction set contains support for byte and word load, store, and byte-manipulating instructions. The individual instructions are listed in Table A.9 and are described in the following paragraphs.

The LDxU instructions load a byte or word from the specified address in memory and store the zero-extended result in the destination register. Note that the normal LDL instruction sign-extends the result instead. The STx instructions store a byte or word to the specified address in memory.

There is a rich set of byte manipulation instructions: The CMPBGE instruction compares corresponding bytes of the two quadword source operands in parallel and stores the outcome of all comparisons in the least significant byte of the destination register. Starting at an arbitrary byte location within the source quadword, the EXTxx instructions extract a byte, word, longword, or quadword, respectively. The vacated bit positions are zero-filled. The difference between the EXTxL and EXTxH forms is the reference point for the byte boundary: The EXTxL instructions count bytes starting with the least significant byte, while the EXTxH instructions count bytes starting with the most significant byte.

The INSxx instructions perform the reverse operation: a byte, word, longword, or quadword from the source operand is inserted at an arbitrary byte position within the target quadword, filling the vacated bit positions with zero. The MSKxx instructions insert a zero byte, word, longword, or quadword at an arbitrary byte position. The ZAP/ZAPNOT instruction pair per-

Table A.9. Byte & Word Extension Instructions

Name	Description
CMPBGE	Byte Vector Compare
EXTBL	Extract Byte Low
EXTWL	Extract Word Low
EXTLL	Extract Longword Low
EXTQL	Extract Quadword Low
EXTWH	Extract Word High
EXTLH	Extract Longword High
EXTQH	Extract Quadword High
INSBL	Insert Byte Low
INSWL	Insert Word Low
INSLL	Insert Longword Low
INSQL	Insert Quadword Low
INSWH	Insert Word High
INSLH	Insert Longword High
INSQH	Insert Quadword High
LDBU	Load Byte Unsigned
LDWU	Load Word Unsigned
MSKBL	Mask Byte Low
MSKWL	Mask Word Low
MSKLL	Mask Longword Low
MSKQL	Mask Quadword Low
MSKWH	Mask Word High
MSKLH	Mask Longword High
MSKQH	Mask Quadword High
SEXTB	Sign-Extend Byte
SEXTW	Sign-Extend Word
STB	Store Byte
STW	Store Word
ZAP	Zero Bytes
ZAPNOT	Zero Bytes Not

forms a similar operation, but allows arbitrary, e.g. non-contiguous, bytes in a quadword to be filled with zeros. The SEXTx instructions provide sign-extension for byte and word operands. These instructions are necessary since the byte and word load instructions perform zero-extension instead of sign-extension.

Multimedia extension (MVI). Instructions in the multimedia extension are targeted at audio and video algorithms, e.g. MPEG compression and decompression, hence the name (motional video instructions). The individual instructions in this extension are listed in Table A.10 and are described in the following paragraphs.

The MINxB8/MINxW4 instructions compare the corresponding byte or words of the two source quadwords in parallel and store the minimum value for each comparison in the destination register. The MINSxx instructions perform signed comparisons, while the MINUxx instructions perform unsigned comparisons. The MAXxxx instructions are similar, but store the maximum

Table A.10. Multimedia Extension Instructions

Name	Description
MINUB8	Byte Vector Unsigned Minimum
MINSB8	Byte Vector Signed Minimum
MINUW4	Word Vector Unsigned Minimum
MINSW4	Word Vector Signed Minimum
MAXUB8	Byte Vector Unsigned Maximum
MAXSB8	Byte Vector Signed Maximum
MAXUW4	Word Vector Unsigned Maximum
MAXSW4	Word Vector Signed Maximum
PERR	Pixel Error
PKLB	Pack Longwords to Bytes
PKWB	Pack Words to Bytes
UNPKBL	Unpack Bytes to Longwords
UNPKBW	Unpack Bytes to Words

value for each comparison in the destination register. These instructions can be used to perform clamping to maximum or minimum values.

The PKxB instructions truncate two longwords or four words of the source operand to bytes and store these two or four bytes in the least significant two/four byte locations of the target operand. The reverse operation is performed by the UNPKxx instructions. The PERR instruction is useful for motion estimation and performs a comparison of corresponding bytes in the two source operands in parallel and returns the sum of the differences.

Floating-Point extension (FIX). The floating-point extension contains instructions that transfer data between integer and floating-point registers without accessing memory, as well as the square root arithmetic instruction. The individual instructions are listed in Table A.11 and are described in the following paragraph.

The SQRTx instructions provide the square-root operation for all supported floating-point data types. The FTOIx and ITOFx instructions transfer data between integer and floating-point registers (ITOFx) and vice versa (FTOIx). The instructions do not interpret the contents of the registers in any way, hence no arithmetic exceptions can occur. The CVTxx instructions should be used to convert between integer and floating-point data types.

Count extension (CIX). The count extension is the latest addition to the Alpha architecture instruction set and contains the three instruction listed in Table A.12. The CTLZ instruction returns the number of leading zeros, i.e. the number of zeros starting from the most significant bit downwards until the first one is encountered, to the destination register. The CTTZ instruction performs the same operation starting from the least significant bit upwards. The CTPOP instructions returns the number of ones in the source operand to the destination register. All instructions operate on quadwords only and generate no exceptions.

Table A.11. Floating-Point Extension Instructions

Name	Description
FTOIS	Floating-Point to Integer Register Move, Sfloat
FTOIT	Floating-Point to Integer Register Move, Tfloat
ITOFF	Integer to Floating-Point Register Move, Ffloat
ITOFS	Integer to Floating-Point Register Move, Sfloat
ITOFT	Integer to Floating-Point Register Move, Tfloat
SQRTF	Square Root, Ffloat
SQRTG	Square Root, Gfloat
SQRTS	Square Root, Sfloat
SQRTT	Square Root, Tfloat

Table A.12. Count Extension Instructions

Name	Description
CTLZ	Count Leading Zero
CTPOP	Count Population
CTTZ	Count Trailing Zero

A.2.4 PALcode

PALcode is used to provide a consistent interface for operating systems across different implementations. The individual routines depend on the operating system, but typically include operations such as memory management, interrupt and exception handling as well as context switching. PALcode resides in memory and is written in standard machine code with some notable extensions: PALcode is executed in an environment that provides full control over the hardware, access to additional state, and runs in the absence of interrupts or page faults. Since the PALcode is operating-system dependent, it can be switched at boot-time to support different operating systems.

A.3 Implementations

The 21064 microprocessor introduced in 1992 was the first implementation of the Alpha architecture. It was soon followed by several derivatives: the 21064A, 21066, 21066A, and 21068 microprocessors. These processors represent the first generation of Alpha architecture implementations. The 21164 microprocessor introduced in late 1994 was the first implementation of the second-generation, again followed by several derivatives: the 21164A and 21164PC microprocessors. These processors represent the second generation of Alpha architecture implementations. The first of the third-generation implementations, the 21264 microprocessor, was introduced in 1998 and was followed by the 21264A and 21264B microprocessors. These microprocessors represent the third-generation of Alpha architecture implementations. According to the current road-maps, the first fourth-generation implementation

will enter the marketplace in late 2001. Unfortunately, this implementation will probably be the last implementation of the Alpha architecture, as the Alpha architecture has recently been discontinued.

The following sections discuss the first-, second-, and third-generations of implementations in detail and give an outlook on the fourth- as well as the recently abandoned fifth-generation implementations. For information about the corresponding systems, which is outside the scope of this chapter, the reader is referred to issues of the Digital/Compaq Technical Journal.

A.3.1 Alpha 21064

The 21064 microprocessor code-named EV4 was introduced in 1992 and represents the first full[1] implementation of the Alpha architecture. The chip operates at up to 200 MHz, executing up to 400 million instructions per second since it can issue up to two instructions in parallel. The 21064 does support the basic instruction set and the IEEE and VAX floating-point subsets as defined by the Alpha architecture, i.e. does not support any of the instruction set extensions. The memory system includes two 8 KB data and instruction caches as well as support for an external second-level cache. The 21064 supports a virtual and physical address size of 43 and 34 bits, respectively.

The chip contains 1.68 million transistors and was fabricated in a 0.7 µ 3-layer metal CMOS process, yielding a die size of 12.4×15.0 mm. Using a supply voltage of 3.3 V, the chip dissipates up to 30 W at 200 MHz. The 21064 is packaged in a ceramic pin grid array (CPGA) package with 431 pins. The internal architecture of the 21064 is sketched in Figure A.4, the following sections describe the individual elements in more detail. The information presented in these sections was gathered from the corresponding hardware reference manual [AXP96a] as well as several articles [DWA+92a][DWA+92b].

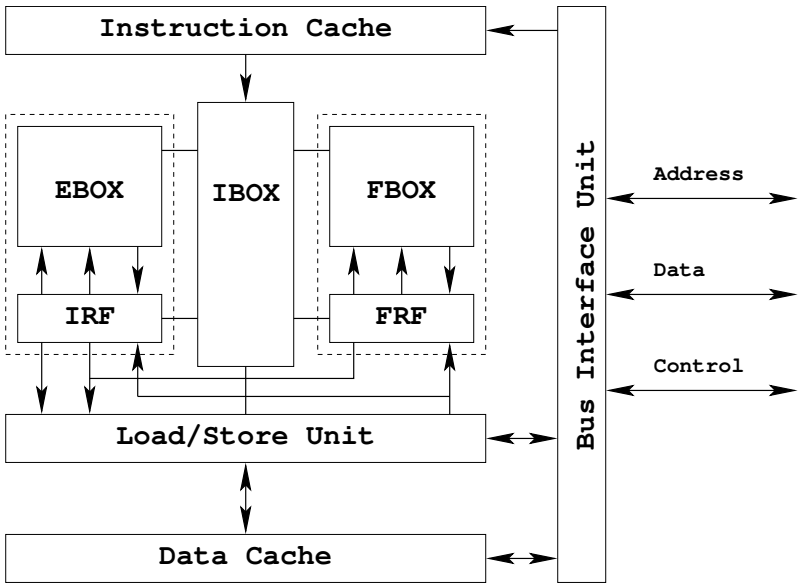
Instruction Cache. The instruction cache is an 8 KB direct-mapped cache that uses 32 byte cache lines. Apart from the tag and data fields, each cache-line contains an 8 bit branch history field that is used for branch prediction. The cache is virtually indexed and tagged, since the size of the cache is equal to the page size. There is no mechanism for maintaining coherence with memory, i.e. the software is responsible to issue the IMB instruction after the instruction stream has been modified.

An interesting feature is the stream buffer: On a cache miss, the cache fetches the missing cache-line from memory, but loads the next cache-line into the stream buffer as well. If this line is subsequently requested by the instruction fetch and decode unit, the whole cache-line is transfered to the instruction cache in a single cycle.

The translation between virtual and physical addresses for the instruction stream is performed by two instruction translation buffers: one fully-associative buffer with eight entries for 8 KB pages and another fully-

[1] There was an earlier integer-only implementation for internal use

Fig. A.4. Alpha 21064 Internal Architecture



associative buffer with four entries for 4 MB pages. These instruction translation buffers are maintained by PALcode.

Instruction Fetch and Decode Unit. The instruction fetch and decode unit (IBOX) has three primary tasks, i.e. to fetch, decode, and issue instructions. Apart from these tasks, the instruction fetch and decode unit is responsible for controlling the execution pipelines, especially in the presence of exceptions, traps, or interrupts. Instructions are fetched from the instruction cache by the prefetcher. The prefetcher generates a quadword address for the next access to the instruction cache in each cycle, i.e. fetches two instructions simultaneously.

Branch prediction is used to avoid pipeline stalls due to discontinuities in the instruction stream: The instruction cache stores a single bit of branch history for every branch instruction. This history bit is used to predict whether the branch is taken or not-taken on the next execution. If the branch has never been executed before, static branch prediction is used: backward branches are predicted taken, while forward branches are predicted not-taken. In addition to the branch history table, the instruction fetch and decode unit contains a four-entry return address stack to be used by indirect branch instructions.

After an instruction pair has been fetched, both instructions are decoded and the availability of the required resources is checked in parallel. Instructions are issued in-order, i.e. the second instruction does not issue until the first instruction can be issued. If the resources for both instructions are avail-

able and certain conditions are met, both instructions can be issued in parallel: a load and store instruction with an integer of floating-point operate instruction, an integer operate instruction together with a floating-point operate instruction, or a branch instruction together with a load/store, integer or floating-point operate instruction.

There are two exception to the dual-issue rules above: integer branches and integer stores as well as floating-point branches and floating-point stores can not be issued together as well as integer stores and branches together with floating-point operate instructions and floating-point stores and branches together with integer operate instructions. These restrictions reflect the number of read and write ports for the integer and floating-point register files.

Note that dual-issue is only possible if both instructions come from the same instruction pair, i.e. if there is only one instruction from the current instruction pair, the prefetcher does not try to dual-issue the remaining instruction with the first instruction from the next instruction pair. Apart from the dual-issue rules mentioned above, there are several other issue rules. The interested reader is referred to the corresponding Hardware Reference Manual [AXP96a].

Integer Unit. The integer unit consists of the integer execution unit (IBOX) and the integer register file (IRF). The integer register file contains 32 registers, each 64 bits wide, as defined by the Alpha architecture. The integer register file has four read and two write ports: two read and one write port dedicated to the integer execution unit as well as two read and one write port shared between the instruction fetch and decode unit and the load/store unit.

The integer execution unit contains several fully pipelined units: a 64 bit arithmetic-logic unit, a barrel shifter, and a multiplier. All integer arithmetic instructions except multiply have a latency of one or two cycles. The multiplier retires four bits per cycle, yielding a latency of 21 and 23 cycles for longword and quadword operands, respectively.

Floating-Point Unit. The Floating-Point Unit consists of the floating-point execution unit (FBOX) and the floating-point register file (FRF). The floating-point register file contains 32 floating-point registers, each 64 bits wide, as defined by the Alpha architecture. The floating-point register file has three read and two write ports: two read and one write port dedicated to the floating-point execution unit as well as one read and one write port shared between the instruction fetch and decode unit and the load/store unit. Note that a second read port for the latter units is not necessary, since address operands are always stored in an integer register.

The floating-point execution unit supports the IEEE as well as the VAX subsets, with the exception of the IEEE $+\infty$ and $-\infty$ rounding modes and the inexact flag for divide operations. These operations must be performed in software. The floating-point execution unit contains an adder/multiplier that has a latency of six cycles and is fully pipelined, i.e. can accept new operands in every cycle. Division is handled by a separate non-pipelined divider that

can retire a single bit in every clock cycle. The latency for divide operations is therefore up to 34 and 63 cycles for single-precision and double-precision operands, respectively.

Load/Store Unit. The load/store unit (ABOX) provides the execution core with an interface to the data cache and the external bus interface unit. The load/store unit consists of an address generator, a write buffer, and a load silo. The address generator is responsible for calculating the effective virtual address for load and store instructions by adding the sign-extended displacement to the contents of the base register.

The write buffer decouples the execution units from the bus interface unit. Since the core can generate store instructions at a higher rate than the external cache or main memory can accept them, this decoupling is necessary in order to avoid pipeline stalls. To this end, the write buffer contains a four-entry queue, such that each entry contains a whole cache-line (32 bytes). New stores are appended to the end of the queue, while stores at the head of the queue are sent to the bus interface unit if certain conditions are met.

Instead of simply buffering the stores, the write buffer allows stores to the same cache-line to merge with already existing entries. Note that the merging of stores may change the number as well as the ordering of stores - memory barriers must be inserted in the instruction stream if strict ordering is required. Apart from this reordering by merging, stores are not reordered.

The load/store unit can accept new commands until a data cache fill is required, i.e. until a load miss occurs, since the data cache does not use a write-allocate protocol. In the case of a load miss, instructions using the load/store unit are no longer issued until the load miss has been resolved. Since load misses are detected rather late in the pipeline, there may be up to two instructions destined for the load/store unit in the pipeline. These load/store instructions are placed in silos until the first load miss is resolved and are subsequently replayed in program order. An exception to this rule are loads that hit in the data cache, these are allowed to complete even in the case of an outstanding cache fill.

Data Cache. The data cache is an 8 KB direct-mapped cache that uses 32 byte cache-lines. The cache is virtually indexed and tagged since the size of the cache is equal to the page size. Load instructions that hit in the cache incur a latency of three cycles. A write-through protocol is used to update external caches or main memory. The data cache is kept coherent with main memory by an invalidate bus, i.e. external system logic is responsible for cache coherence. An interesting feature is the pending fill latch: Incoming data from previous fill requests is accumulated in this latch while the cache processes other requests, i.e. the cache supports hit-under-miss. If a whole cache-line has accumulated in the pending fill latch, the whole cache-line is written to the cache in a single cycle.

The translation between virtual and physical addresses for the data stream is performed by the data translation buffer, a fully-associative buffer

with 32 entries for up to 512 pages per entry, each 8 KB large. Like the instruction translation buffer, the data translation buffer is maintained by PALcode.

Bus Interface Unit. The bus interface unit connects the internal caches as well as the load/store unit to the external bus interface. A fixed priority scheme is used to schedule requests from these three sources: data cache fills have the highest priority, instruction cache fills come next, while the load/store unit has the lowest priority. In addition, the bus interface unit supports an external direct-mapped cache that ranges from 128 KB to 16 MB in size.

A.3.2 Alpha 21064A

The 21064A microprocessor, code-named EV45, was introduced in late 1993 and is very similar to the 21064 described above. However, several enhancements were made to the original design: The size of the instruction and data caches was doubled to 16 KB each. In addition, error-correcting codes (ECC) were added to both caches. The branch prediction was improved as well, as the instruction cache maintains a two-bit saturating counter for each instruction location. The floating-point execution unit was updated with a new division algorithm that is able to retire 2.4 bits (on average) per cycle instead of the original single bit per cycle. This reduces the latency for division operations to 15 to 31 (19 average) and 22 to 60 (31 average) cycles for single-precision and double-precision operands, respectively. The divider also calculates the inexact flag correctly, therefore traps to software are no longer necessary to calculate this flag.

The 21064A contains 2.8 M transistors due to the larger caches. The chip was fabricated in an improved 0.5 µ 4-layer metal CMOS process, yielding a die size of $10.5 \times 14.5\,\mathrm{mm}^2$ compared to the original $12.4 \times 15.0\,\mathrm{mm}^2$. These process improvements allow the 21064A to operate at a clock frequency of 275 MHz. Using a supply voltage of 3.3 V, the chip dissipates up to 33 W. The 21064A is packaged in the same 431 CPGA package like the 21064. In addition, the 21064A uses the same pin assignment. More information on this implementation can be found in the corresponding hardware reference manual [AXP96a].

A.3.3 Alpha 21066

The 21066 microprocessor, code-named LCA4 (Low Cost Alpha), is a derivative of the earlier 21064. The 21066 integrates several new functions around a 21064-based core: a Peripheral Component Interface (PCI) bus interface, a cache controller, a memory controller, a simple graphics controller, and a phased-locked loop. The core is almost identical to the 21064, but the external data bus width was decreased from 128 to 64 bits.

Due to the additional functions, the 21066 contains 1.75 M, i.e. slightly more than the original 21064. The chip was fabricated in a 0.675µ 3-layer metal CMOS process, yielding a die size of $12.3 \times 17.0\,\text{mm}^2$. The maximum operating frequency was reduced to 166 MHz in order to increase yields and lower cost. Unfortunately, the lower clock frequency together with the reduced size of the data bus caused a significant performance loss compared to the 21064, hence the chip was primarily used in embedded applications. More information on this implementation can be found in the corresponding hardware reference manual [AXP96b] or Digital Technical Journal [MBC+94].

A.3.4 Alpha 21068

The 21068 microprocessor, code-named LCA4S, is identical to the 21066 microprocessor, but the maximum clock frequency is specified as 66 MHz, sometimes 100 MHz. The chip was fabricated by Samsung, hence the capital S in the internal designation.

A.3.5 Alpha 21066A

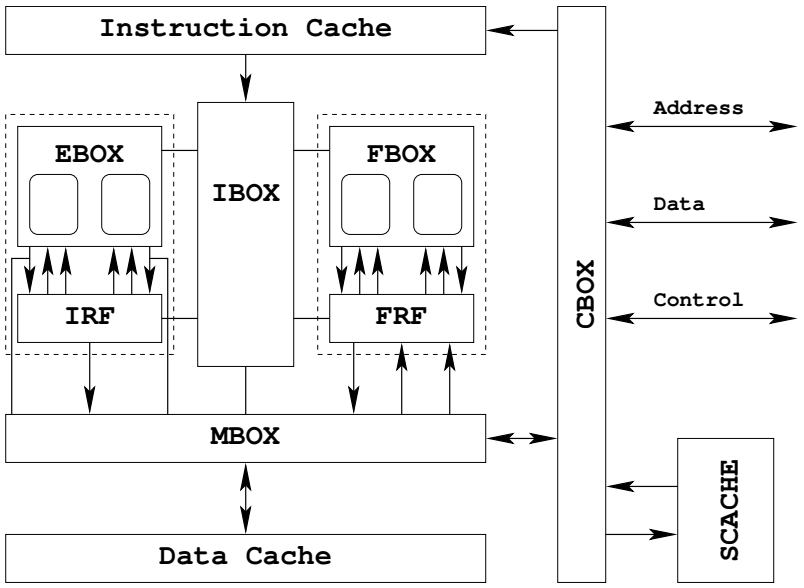
The 21066A microprocessor, code-named LCA45, is based on the earlier 21066 microprocessor, but contains some minor improvements: The 21066A uses the advanced floating-point divide algorithm introduced in the 21064A and was fabricated in an improved CMOS process. The 21066A contains 1.8 M transistors and was fabricated in a 0.5µ 3-layer metal CMOS process, yielding a die size of $10.9 \times 14.8\,\text{mm}^2$. The maximum clock frequency was increased to 233 MHz. More information on this implementation can be found in the corresponding hardware reference manual [AXP96b].

A.3.6 Alpha 21164

The 21164 Microprocessor, code-named EV5, was introduced in late 1994 and represents the first second-generation implementation of the Alpha architecture. The chip executes up to 1.2 billion instructions per second due to the capability to issue up to 4 instructions in each clock cycle and the 300 MHz clock frequency. The 21164 supports the byte and word extension of the Alpha architecture instruction set as well as the IEEE and VAX floating-point subsets. The memory system includes two 8 KB data and instruction caches, an internal 96 KB unified second-level cache, as well as support for an external third-level cache. The 21164 supports a virtual and physical address size of 43 and 40 bits, respectively.

The chip contains 9.3 M transistors and was fabricated in a 0.5 µ 4-layer metal CMOS process, yielding a die size of $16.5 \times 18.1\,\text{mm}^2$. Using a supply voltage of 3.3 V, the chip dissipates up to 50 W. Like the earlier implementations, the 21164 is housed in a CPGA package, but the number of pins was

Fig. A.5. Alpha 21164 Internal Architecture



increased to 499. The internal architecture of the 21164 is depicted in Fig. A.5, the following sections describe the individual elements in detail. The information presented in these sections was gathered from the corresponding hardware reference manual [AXP96c] and several journal articles [ERPR95], [ERB+95], [KN95].

Instruction Cache. The instruction cache is an 8 KB direct-mapped cache with 32 byte cache-lines. The cache has a single read and a single write port. Cache coherency must be maintained in software by issuing IMB instructions. Apart from the data and tag arrays, each cache-line contains a branch history table using two-bit saturating counters for each instruction location, similar to the instruction cache in the earlier 21064A processor. To improve yields, the instruction cache has redundant rows that can be used to replace defective rows during wafer probe.

Instruction Fetch and Decode Unit. The instruction fetch and decode unit (IBOX) has five primary tasks: to fetch, decode, slot, and issue instructions, and to predict branches. Apart from these tasks, the instruction fetch and decode unit is responsible for controlling the execution pipelines, especially in the presence of exceptions, traps, or interrupts. The following paragraphs describe the individual tasks in detail.

Instruction fetch is performed by the prefetcher that fetches an aligned block of four instructions from the instruction cache in each cycle. The instructions in the instruction cache are already partially decoded to ease the

task of the instruction fetch and decode unit. Afterwards the instructions are decoded in parallel and stored in one of the two instruction buffers. Each instruction buffer is capable to hold four decoded instructions. Branch prediction is used to determine the next fetch address and the instruction translation buffer is accessed.

The instruction cache stores two bits of branch history for every branch instruction. These history bits are used to predict whether the branch is taken or not-taken on the next execution. If the branch has never been executed before, the same static branch prediction as in the 21064 is used. In addition to the branch history table, the instruction fetch and decode unit contains a four-entry return address stack to be used by indirect branches. Note that the branch prediction logic is capable to predict up to six branches in each cycle.

Instruction slotting resolves all static conflicts and assigns the individual instructions to appropriate execution pipelines, while dynamic conflicts are handled in the issue stage. The slotting stage is able to slot all four instruction in parallel for instruction mixes that contain one instruction for each execution pipeline. The instruction fetch and decode unit slots instructions in program order, since the 21164 issues instructions in program order. The slotting logic assigns each instruction to one of the four execution pipelines until it encounters an instruction for which no execution pipeline is available. Integer instructions that can be executed by both integer execution pipelines are assigned to the first integer pipeline, unless this pipeline has already been allocated or the next instruction can only be allocated to this pipeline. A similar rule applies for floating-point instructions that can execute in either the add or the multiply pipelines. These instructions are assigned to the add pipeline unless the add pipeline has already been allocated. Apart from the rules mentioned above, the slotting logic enforces other rules, which are described in detail in the corresponding hardware reference manual [AXP96c].

The slotted instructions advance to the issue stage, while the remaining instructions will be slotted in subsequent cycles. Only instructions within the original group of four instructions are slotted together, i.e. the next group of instructions enters the slotting stage only if all instructions from the previous group have advanced to the issue stage. The issue stage checks instructions for dynamic conflicts, e.g. operand and resource conflicts. The instructions are issued in program order, i.e. instruction issue stops whenever an instruction with remaining conflicts is encountered, and resumes after these conflicts have been resolved. The issue logic is capable to issue up to four instructions in each cycle in the absence of dynamic conflicts.

Integer Unit. The integer unit consists of an execution unit (EBOX) as well as a register file (IRF). The integer execution unit contains two independent pipelines: the first pipeline is able to handle all arithmetic and logic instructions including multiplication, as well as load, store, branch, and jump instructions. The second pipeline can handle all arithmetic and logic instruc-

tions with the exception of multiplication, but cannot handle stores, branch, or jump instructions. Note that load instructions can be handled by both pipelines. All integer arithmetic instructions except multiply and conditional moves have a latency of one cycle. Conditional moves have a latency of two cycles, while the multiplier has a latency of 8 and 16 cycles for longword and quadword operands, respectively.

The integer register file contains 40 registers, each 64 bits wide: the 32 integer registers defined by the Alpha architecture as well as 8 additional registers that are only accessible to PALcode. The register file has four read and two write ports: two read and one write port for each of the execution pipelines. Note that the write ports are shared between the execution units and the memory subsystem, i.e. the memory address translation unit.

Floating-Point Unit. The floating-point unit consists of the floating-point execution unit (FBOX) as well as the floating-point register file (FRF). The execution unit contains two separate pipelines instead of the single pipeline used in earlier implementations. One of the pipelines executes multiply instructions, while the other pipeline executes all other instructions. The floating-point unit is therefore able to process up to two instructions in each clock cycle.

The floating-point register file supports the 32 registers required by the Alpha architecture, each 64 bits wide. The register file has five read ports as well as four write ports: two read and one write port for each execution pipeline, one read port and two write ports for the memory address translation unit.

Compared to the 21064 the latency of floating-point operations was reduced to four cycles for all operations except for division. The non-pipelined divider uses the same algorithm that was used in the 21064A processor, hence the latency for divide operations is 15 to 31 (19 average) and 22 to 60 (31 average) cycles for single-precision and double-precision operands, respectively. The divider is associated with the add pipeline, instructions can still be issued to the add pipeline even if a division is in progress. The floating-point unit supports all IEEE and VAX rounding modes in hardware, including the $+\infty$ mode that was left out in earlier implementations.

Memory Address Translation Unit. The memory address translation unit (MBOX) provides the interface between the execution units and the memory system. The unit contains data and instruction translation buffers, a miss address file, as well as an write buffer. Note the absence of an address generation unit present in earlier implementations, this task is now performed by the integer execution unit, i.e. the memory address translation unit already receives virtual addresses from the integer unit.

The data translation buffer is a 64-entry, fully associative memory that stores the virtual to physical address mappings for different page sizes: Each entry supports up to 512 pages, each 8 KB large. A not-last-used strategy is used to replace older entries. Compared to the 21064, the data translation

buffer is considerably larger and has two read/write ports, since up to two virtual addresses must be translated in each cycle.

The instruction translation buffer is a 48-entry, fully associative memory that stores the virtual to physical address mappings for different page sizes: Each entry supports up to 512 pages, each 8 KB large. A not-last-used strategy is used to replace older entries. Compared to the data translation buffer, the instruction translation buffer has only one read/write port.

The miss address file has ten entries and stores the address and data information of loads that miss in the caches. Subsequent load misses check the miss address file and are merged with existing entries under certain conditions. This allows multiple load misses to be serviced with a single fill from the cache control and bus interface unit. The miss address file has six entries for loads as well as four entries for instruction fetches. Each entry stores the address of a 32 byte cache-line.

The write buffer is similar to the one used in the 21064, but has increased to six entries. Note that the peak rate for stores is still one per cycle, since stores can only be executed by the first integer execution pipeline.

Data Cache. The data cache is an 8 KB direct-mapped cache with 32 byte cache-lines. The cache has two read ports as well as one write port and uses a write-through, read-allocate protocol. Load instructions that hit in the cache incur a latency of two cycles. The data cache is duplicated for performance reasons, both copies are written simultaneously. The cache coherency is maintained by the cache control and bus interface unit described below. In order to improve yields, the data cache has redundant rows that can be used to replace defective rows during wafer probe.

Secondary Cache. The internal second-level cache is a 96 KB, 3-way set-associative cache with a selectable cache-line size of 32 or 64 bytes. The cache is unified for data and instructions and uses a write-back, write-allocate protocol. Load instructions that hit in the cache incur a latency of eight or more cycles. Up to two outstanding cache requests can be queued. The secondary cache has redundant rows and columns that can be used to replace defective rows or columns during wafer probe in order to improve yields.

Cache Control and Bus Interface Unit. The cache control and bus interface unit connects the memory subsystem to the internal second-level cache, the external third-level cache as well as the system bus interface. The unit controls the second- and third-level caches and is responsible for maintaining cache coherency by implementing a modify, exclusive, shared, invalid (MESI) protocol for multiprocessor systems.

The external third-level cache is a direct-mapped cache ranging from 1 to 64 MB in size with a selectable cache-line size of 32 or 64 bytes. Like the internal second-level cache, the third-level cache uses a write-back, write-allocate protocol.

A.3.7 Alpha 21164A

The 21164A, code-named EV56, is very similar to the 21164 described above, but is fabricated in an improved 0.35 µ 4-layer metal CMOS process. The 21164A uses the same number of transistors like the 21164, i.e. 9.67 M, which yields a die size of 299 mm^2 in the improved process. The maximum operating frequency was increased to 625 MHz, at which the chip dissipates a maximum heat of 62 W. More information on this implementation can be found in the corresponding hardware reference manual [AXP98a].

A.3.8 Alpha 21164PC

The 21164PC, code-named PCA56 (Personal Computer Alpha), was introduced in 1997 and is based on the 21164. Several modifications were made in order to lower the cost of the chip: The internal second-level cache was omitted, the choice of cache coherency protocols was restricted to the flush-based coherency protocol, and the physical address size was reduced to 33 bits. Several improvements were made as well: The 21164PC supports the MVI extension of the Alpha architecture and the size of the internal instruction cache was increased to 16 KB.

The chip consists of 3.5 M transistors and was fabricated by Mitsubishi in a 0.35 µ 4-layer metal CMOS process, yielding a die size of $8.65 \times 16.28\,mm^2$. The 21164PC has a maximum operating frequency of 533 MHz and dissipates up to 32 W using a core supply voltage of 2.5 V. More information on this implementation can be found in the corresponding hardware reference manual [AXP97].

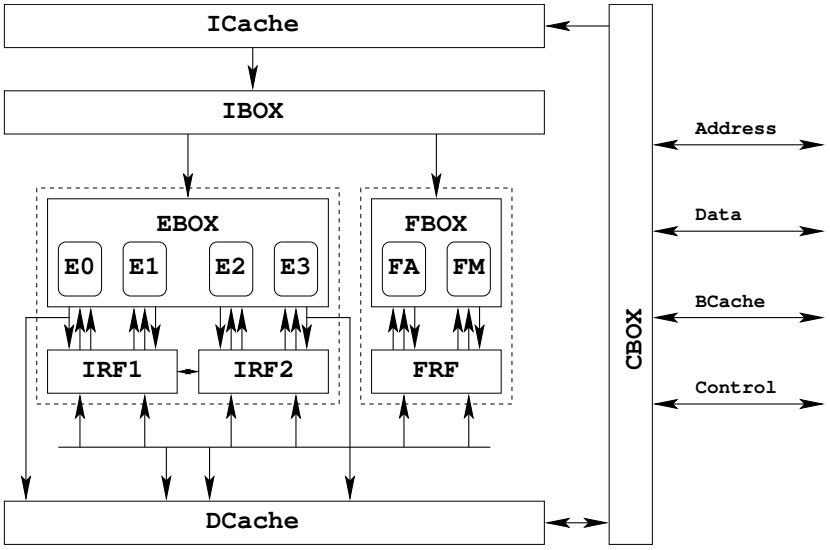
The 21164PC code-named PCA57 is similar to the PCA56 described above, but the design was further improved: The size of the internal instruction cache was increased to 32 KB, and the organization is 2-way set-associative instead of direct-mapped. The size of the internal data cache was increased to 16 KB as well. The number of entries in the write buffer was increased from six to eight. The floating-point execution unit was improved in order to reduce the latency of floating-point multiplies and divisions: A multiply instruction has a latency of three cycles instead of the original four cycles. The division algorithm retires 6 bits in each cycle compared to the original 2.4 bits.

The PCA57 consists of 7 M transistors and was fabricated in a 0.28 µ 4-layer metal CMOS process, yielding a die size of $6.7 \times 15\,mm^2$. The chip has a maximum operating frequency of 583 MHz. More information on this implementation can be found in the corresponding hardware reference manual [AXP98b].

A.3.9 Alpha 21264

The 21264 processor, code-named EV6, was introduced in 1998 and represents the first third-generation implementation of the Alpha architecture.

Fig. A.6. Alpha 21264 Internal Architecture



The processor uses advanced techniques like out-of-order issue and is able to execute up to six instructions per clock cycle. The memory system is different from earlier implementations as it uses two 64 KB instruction and data caches as well as an external second-level cache. The 21264 supports the BWX, MVI, and FIX extensions to the Alpha architecture instruction set as well as the IEEE and VAX floating-point subsets. The 21264 supports a virtual and physical address size of 48 and 44 bits, respectively.

The chip consists of 15 M transistors and is fabricated in a 0.35 µ 6-layer metal CMOS process, yielding a die size of 310 mm^2. Using a core supply voltage of 2.2 V, the chip dissipates up to 60 W at the maximum clock frequency of 500 MHz. The 21264 is packaged in a pin grid array (PGA) with 587 pins.

The internal architecture of the 21264 is depicted in Fig. A.6, the following sections describe the major elements in detail: instruction cache, instruction fetch/decode unit, integer unit, floating-point unit, data cache, and cache control unit. The information in these sections was gathered from various sources, namely the corresponding hardware reference manual [AXP00a] and several articles [Gwe96][LR97][MBB+98][KMW98][Kes99].

Instruction Cache. The instruction cache (ICache) is a 64 KB two-way set-associative cache that uses a cache-line size of 64 bytes. The cache is virtually indexed, physically tagged, and protected by error-correcting codes. In each cycle, the instruction cache delivers up to four instruction to the instruction fetch and decode unit. In order to reduce fetch bubbles, the instruction cache

uses two additional fields in each cache-line to predict the line and set of the next instruction cache access. The line and way prediction fields are trained by the instruction fetch and decode unit, i.e. the branch prediction logic. An additional two-bit saturating counter ensures that the prediction fields are only trained when multiple wrong predictions in a row occur.

Instruction Fetch and Decode Unit. The instruction fetch and decode unit (IBOX) has five primary tasks: fetching, decoding, issuing, and retiring instructions as well as branch prediction. Apart from these tasks, the instruction fetch and decode unit is responsible for controlling the execution pipelines, especially in the presence of exceptions, traps, or interrupts. The unit consists of the program counter, the branch prediction, register renaming, as well as retiring logic. The program counter logic maintains the addresses of all instructions that are in-flight, i.e. are currently executing. The instruction fetch addresses are stored in a 20-entry queue, hence there can be up to 80 in-flight instructions, as each instruction fetch consists of four instructions.

The branch prediction logic was significantly enhanced from earlier implementations of the Alpha architecture: The 21264 processor uses a combination of three different branch predictors, i.e. the local, global, and choice predictors.

The local predictor uses a two-level table to store branch history information. The first level consists of a branch history table with 1024 entries that store the history of the ten most recent branches. This table is indexed with the lower bits of the virtual address of the branch instruction. Note that different branches may share the same entry if their virtual addresses are identical in these low-order bits. The second level consists of a 1024-entry branch prediction table that is indexed with the results from the first level. Each entry in the second-level table contains a three-bit saturating counter that is used to predict the outcome (taken/not-taken) of the branch.

The global predictor uses a 12-entry global history buffer to store the outcome of the 12 most recent branches. The contents of this buffer are used to index a 4096-entry branch prediction table. Each entry contains a two-bit saturating counter that is used to predict the outcome (taken/not-taken) of the branch.

The choice predictor selects between the local and global predictors. The contents of the global history buffer described above are used to index a 4096-entry branch prediction table. Each entry contains a two-bit saturating counter that is used to select between the predictions from the local or global branch predictor.

The register renaming logic maps the architected registers specified by the instructions to the physical registers of the 21264 processor. The integer and floating-point register files each support 72 registers instead of the 32 registers defined by the architecture. The register mapping is used to resolve write-after-read and write-after-write dependencies as well as allowing speculative

instruction execution. The mapping logic is able to map four instructions in each cycle and stores the complete register mapping for each instruction in an 80-entry buffer. Note that the number of entries corresponds to the number of in-flight instructions. These register mappings are used in the case of branch mispredictions and exceptions, such that the register mapping that corresponds to the offending instruction can be restored. The renaming logic is able to restore the map of an instruction in a single cycle.

After renaming, the instructions are forwarded to the integer and floating-point issue queues. In order to maintain the sequential programming semantics, instructions are retired in program order, although they are issued and executed out-of-order. An instruction is retired if all previous instructions have been executed without causing exceptions, e.g. branch mispredictions. The retiring logic is able to retire up to 11 instruction in a single cycle and can sustain a retire rate of eight instructions per cycle.

Integer Unit. The integer unit consists of the integer instruction queue, the four integer execution pipelines (EBOX) organized in two clusters, as well as one copy of the integer register file (IRF1, IRF2) for each cluster. Each copy of the register file contains 80 registers, each 64 bits wide: These registers are used for register renaming of the architected 32 registers as well as 8 registers that are reserved for PALcode. The integer register file was duplicated due to the large number of read and write ports. Each copy supports six write and four read ports: two read and one write port for each of the two execution pipelines in a cluster, two write ports shared between the floating-point unit and the memory unit, as well as two write ports used to synchronize the two copies of the integer register file. Note that it takes one cycle until results from one cluster are available in the other cluster.

Each cluster consists of a lower and an upper execution pipeline. The lower execution pipelines in both clusters are identical and can execute simple integer operate instructions as well as integer or floating-point load and store instructions. The upper execution pipelines are different: Both pipelines execute simple integer operate instructions as well as branch and shift instructions. However, only one pipeline can execute multiply instructions, while the other pipeline can execute motional video instructions. All integer arithmetic instructions, except multiply, motional video, and count instructions, have a latency of one cycle. Motional video and count instructions have a latency of three cycles. The multiply instruction has a latency of seven cycles, both for longword and quadword operands.

The integer issue queue stores up to 20 instructions and is able to issue up to four instructions in each cycle to the execution pipelines. The instruction fetch and decode unit stores new instructions in the integer issue queue, provided that four or more entries are available. Upon entry, each instruction is statically assigned to one of the two arbiters: one arbiter handles the lower execution pipelines in both clusters, while the other arbiter handles the upper execution pipelines in both clusters. Each arbiter issues the two oldest

instructions in each cycle and slots the instructions to the left or right cluster based upon the position of the instructions in the instruction fetch stream. Note that critical path computations tend to execute on the same cluster due to the additional cycle before results from the other cluster become available.

Floating-Point Unit. The Floating-point unit consists of the floating-point execution unit (FBOX) as well as the floating-point register file (FRF). The execution unit contains two separate execution pipelines similar to the floating-point unit in the 21164 processor. One of the pipelines executes multiply instructions, while the add pipeline executes the remaining instructions. The floating-point unit is therefore able to process up to two instructions in each clock cycle.

The floating-point register file contains 72 registers, each 64 bits wide, that are used for register renaming of the architected 32 registers. The register file has five read ports as well as four write ports: two read and one write port for each execution pipeline, one read port and two write ports for the memory address translation unit.

Compared to the 21164, the latency of floating-point operations was kept at four cycles for all operations except division and square-root. Division operations have a latency of 12 and 15 cycles for single-precision and double-precision operands, while square-root operations have a latency of 18 and 30 cycles for single-precision and double-precision operands, respectively. The non-pipelined divider and square-root units are associated with the add pipeline, but instructions can still be issued to the add pipeline even if a division or square-root instruction is in progress. The floating-point unit supports all IEEE and VAX rounding modes in hardware.

Memory Address Translation Unit. The memory address translation unit (MBOX) controls the internal memory system and can execute any combination of two loads and stores in each cycle. In addition, the unit handles up to 32 outstanding loads, 32 outstanding stores, as well as eight cache invalidations. The memory address translation unit consists of the instruction and data translation buffers, load and store queues, as well as the miss address file.

The instruction translation buffer is a 128-entry fully associative memory that stores the virtual to physical address mappings for different page sizes: Each entry supports up to 512 pages, each 8 KB large. A not-last-used strategy is used to replace older entries. In contrast to the data translation buffer, the instruction translation buffer has only one read/write port.

The data translation buffer is a 128-entry fully associative memory that stores the virtual to physical address mappings for different page sizes: Each entry supports up to 512 pages, each 8 KB large. A not-last-used strategy is used to replace older entries. Compared to the 21064, the data translation buffer is considerably larger and has two read/write ports, since up to two virtual addresses must be translated in each cycle.

The load queue is a 32-entry reorder buffer for outstanding load instructions. The load queue stores load instructions in program order, although the load instructions enter the queue out-of-order. The memory references themselves are stored out-of-order, but loads exit the queue in program order to ensure correct memory behavior.

The store queue is a 32-entry reorder buffer for outstanding store instructions. The store queue buffers store instructions in program order, although the store instructions enter the queue out-of-order. The memory references themselves are stored out-of-order, but stores exit the queue in program order to ensure correct memory behavior.

The load and store queues use dual-ported content-addressable memories to resolve read-after-read, read-after-write, write-after-read, and write-after-write hazards and support the propagation of store data from the store queue to the load queue.

The miss address file has ten entries and stores the address and data information of loads that caused a cache miss. Subsequent load misses check the miss address file and are merged with existing entries under certain conditions. This allows multiple load misses to be serviced with a single fill from the cache control and bus interface unit. The miss address file has six entries for loads as well as four entries for instruction fetches. Each entry stores the address of a 32 byte cache-line.

Data Cache. The data cache is a 64 KB two-way set-associative cache that uses a cache-line size of 64 bytes. The cache is virtually indexed, physically tagged, protected by error correcting codes and uses a write-back, write-allocate protocol. Load instructions that hit in the cache incur a latency of three or four cycles for integer and floating-point loads, respectively. The cache is double-pumped, i.e. operates at two times the core frequency in order to support two independent accesses in each cycle.

Backup Cache. The backup cache is an external, direct-mapped cache, that ranges from 1 to 16 MB in size. The 21264 processor uses a dedicated 128 bit data bus to interface to the backup cache. The maximum bandwidth of 6.4 GB/s across this bus is obtained by using synchronous double-data-rate static random-access memory (DDR SSRAM). The size, speed, and type of the cache memories is selectable to fit a wide range of price/performance points.

Cache Control and Bus Interface Unit. The cache control and bus interface unit provides the interface between the internal memory system and the external cache and system busses. The external cache interface consists of a 128 bit data bus as well as a 20 bit address bus. The bus speed for both point-to-point connections is adjustable, providing a maximum bandwidth of 6.4 GB/s. The system bus is a point-to-point connection that consists of a 64 bit data bus as well as a 45 bit address bus. The speed of these busses is adjustable as well, providing a maximum bandwidth of 3.2 GB/s.

The cache control and bus interface unit consists of the write buffer, the I/O write buffer, the probe queue, as well as a copy of the tag array from the internal data cache.

The write buffer is an eight-entry buffer, where each entry contains a whole cache-line. New stores are appended to the end of the queue, while stores at the head of the queue are sent to the bus interface unit if certain conditions are met. Instead of simply buffering the stores, the write buffer allows stores to the same cache-line to merge with already existing entries.

The I/O write buffer is a four-entry buffer that is used to store I/O write requests from the store queue. The probe queue contains eight entries and is used to store cache probe requests from the system bus.

The cache control and bus interface unit uses a copy of the tag array from the internal data cache to speed up cache fills and probe requests. The 21264 processor supports a rich set of cache probe requests to support a wide range of cache coherency protocols.

A.3.10 Alpha 21264A

The 21264A, code-named EV67, is very similar to the 21264 described above, but is fabricated in an improved 0.25 µ 6-layer metal CMOS process. The 21264A uses the same number of transistors like the 21264, yielding a die size of 225 mm^2 in the improved process. The maximum clock frequency was increased to 750 MHz. Using a supply voltage of 2.1 V the chip dissipates a maximum heat of 90 W. More information on this implementation can be found in the corresponding hardware reference manual [AXP00b].

A.3.11 Alpha 21264B

The 21264B, code-named EV68, is very similar to the 21264 described in Section A.3.9, but is fabricated in an improved 0.18 µ 6-layer metal CMOS process. The 21264B uses the same number of transistors like the 21264, yielding a die size of 115 mm^2 in the improved process. The maximum clock frequency was increased to 1 GHz. Using a supply voltage of 2.1 V, the chip dissipates a maximum heat of 75 W. More information on this implementation can be found in the corresponding hardware reference manual [AXP00c].

A.3.12 Alpha 21364

The 21364 microprocessor, code-named EV7, is expected in 2001/2002 and will represent the first of the forth-generation implementations of the Alpha architecture. The 21364 supports all extensions of the Alpha architecture as well as the IEEE and VAX floating-point subsets. The chip integrates an improved core that is based on the 21264, a large (1.75 MB) unified second-level cache, two multi-channel RDRAM controllers, network routers, and I/O

routers. The network routers offer glue-less support for up to 64 processors in a two-dimensional torus configuration. The information in this section was gathered from several articles [Gwe98][Ban99][MJA+01].

The chip consists of 152 M transistors, the major part (138 M) is consumed by various caches. The chip is fabricated in a 0.18 µ 7-layer metal CMOS process, yielding a die size of $21.1 \times 18.8\,\text{mm}^2$. Note that the routing process was changed from earlier implementations of the Alpha architecture: The 21364 uses routing channels for global busses, while the voltage reference planes were replaced by individual current return lines. Using a core supply voltage of 1.5 V, the chip dissipates up to 120 W at the maximum operating frequency of 1.2 GHz. The 21364 is packaged in a land grid array (LGA) with 1443 pins.

The 21364 core is based on the 21264 core, major changes were only made to the cache controller in order to support the integrated second-level cache. The cache controller supports up to 71 outstanding operations: 16 misses from the first-level caches, 16 victims from the second-level cache, 18 probe requests from external devices, 17 requests from other processors, as well as four I/O requests. The second-level cache is a 1.75 MB seven-way set-associative cache using a write-back, write-allocate policy, as well as ECC protection. The cache can deliver up to 16 bytes in each cycles, providing a bandwidth of 19.2 GB/s.

The two memory controllers interface directly to external Rambus DRAM modules. Each memory controller supports up to four Rambus DRAM channels for a total of eight channels, thus providing 12.8 GB of memory bandwidth. The memory controllers support up to 28 outstanding requests. The network router supports four network links as well as one I/O link, each link provides a bandwidth of 6.4 GB/s. The network links are used to form a two-dimensional torus network with up to 64 processors. The I/O link connects to an external I/O chip set that provides access to various I/O interfaces. The network and I/O router is implemented as an eight-by-seven crossbar, i.e. eight read ports (four network, one I/0, two memory and one cache) and seven write ports (four network, one I/O, two memory). The crossbar uses a distributed arbitration scheme and provides buffering for up to 300 packets.

A.3.13 Alpha 21464

The 21464 processor, code-named EV8, represents the first of the fifth-generation implementations of the Alpha architecture and was scheduled to arrive in 2003/2004. As the Alpha architecture was discontinued recently, the EV8 project was canceled. However, some information about this implementation is already available: The 21464 supports four-way simultaneous multithreading that was discussed in Section 1.4.1, a radical step from earlier implementations [Die99]. The support for simultaneous multithreading was estimated to consume less than 10 % of the transistor budget, as the multithreading support was built on top of a traditional out-of-order core. Apart

from multithreading, the 21464 was planned to have eight function units, a large internal second-level cache (approximately. 3 MB) as well as an integrated multi-channel interface to Rambus DRAM. The 21464 was expected to consist of 250 M transistors and would have been fabricated in a 0.125 μm CMOS process.

B. Cray T3E E-Register Programming

The Cray T3E E-registers are a powerful mechanism to tolerate the latency of accesses to local and remote memory. This goal is achieved by significantly increasing the number of outstanding memory operations and pipelining individual requests. Due to compatibility reasons, the shmem library supports these features only inside function calls, i.e. the hardware is fully utilized only for large block transfers. Only the benchlib supports the full capability of the hardware, but this library was never officially supported and is no longer available to the general public. Therefore direct programming of the E-registers is required to use the full potential of the hardware.

Since the Cray publication [Cra97c] that contains the required programming information is only available under the conditions of an non-disclosure-agreement, the information presented in this chapter is gathered from various public sources instead, most notably the Cray T3E optimization guides [ABH97][Cra97b] and the standard header files hwdefs.h [Cra97a], mpphw.h [Cra98a], and shmem.h [Cra98c].

Section B.1 provides basic information about the E-registers from a programmer's point of view. Section B.2 covers the communication routines used in the implementation of the emulation library for the Cray T3E in detail, thereby covering almost all E-register commands. Section B.3 derives some guidelines for programming the E-registers and describes a serious flaw in early T3Es and the implications for programming the E-registers.

B.1 E-Register Programming

Each processing element has two sets of E-registers: a set of 512 user-accessible E-registers and a set of 128 E-registers reserved for system use. This section will cover only the first set, since access to the second set is prohibited for application programmers. The set of 512 E-registers is mapped to memory starting at offset `MPC_EREG_BASE`.

Each E-Register can be used either as a source and destination E-Register (SADE) or as part of a more-operands block of four E-registers (MOBE). The source-and-destination E-registers are used for the actual data transfers, while the more-operands blocks provide additional information for E-Register commands: The first two E-registers contain the mask and offset for the

hardware centrifuge and are required for every E-Register command. The third E-register usually contains the stride for vector commands, while the fourth E-register usually contains the addend for fetch-and-add commands.

The following conventions are valid for programming the user-accessible set of E-registers:

- E-registers zero to three form a read-only more-operands block that contains a default centrifuge mask, a zero offset, a quadword stride, and a longword addend. This block can be used for most accesses to local or remote memory, thereby eliminating the overhead of initializing a more-operand block prior to each access. These E-registers are located at offset `_MPC_E_REG_STRIDE1` from `MPC_EREG_BASE`.
- E-registers four to seven form a partial read-only more-operands block that contains a default centrifuge mask, a zero offset, and a longword stride. The addend can be set to any value, which is useful for fetch-and-add commands. These E-registers are located at offset `_MPC_E_REG_STRIDE_LW` from `MPC_EREG_BASE`.
- E-registers eight to 15 form two callee-save more-operands blocks. A subroutine may use these E-registers as more-operands blocks only if the contents are saved upon subroutine entry and restored before the subroutine returns. These callee-save blocks are useful if memory arrays with a non-standard distribution are frequently accessed, as the corresponding more-operands blocks do not need to be initialized after each subroutine call. These E-registers are located at offset `_MPC_E_REG_SAV_MOBE` from `MPC_EREG_BASE`.
- E-registers 16 to 32 form four scratch more-operands blocks that may be used without saving and restoring their contents first and afterwards, respectively. In addition, these E-registers can also be used as source-and-destination registers, as long as operations using these E-registers are guaranteed to complete, i.e. never go to full-fault state. These E-registers are located at offset `_MPC_E_REG_SCR_MOBE` from _MPC_E_REG_BASE.
- E-registers 32 to 511 may be used as source-and-destination registers without saving or restoring their contents first or afterwards, respectively. In addition, these E-registers can also be used as more-operands blocks, provided that any full-fault state is cleared first. These E-registers are located at offset `_MPC_E_REG_SADE` from `_MPC_E_REG_BASE`.

Apart from the E-registers themselves, there is a set of state registers for each set of E-registers. Each state register is 64 bits wide and contains the state for 32 E-registers, since the state for a single E-register occupies two bit. As a consequence, there are 16 state registers for the set of user-accessible E-registers, as well as 4 state registers for the set of E-registers reserved for system use. The mapping between E-registers and the corresponding state registers is straight-forward: The ith state register contains the state for the E-registers numbered from $32i$ to $32(i+1)-1$ in ascending order, i.e. the

state for E-register $32i$ is located in the least-significant bits and the state for E-register $32(i+1)-1$ is located in the most-significant bits.

Since the state for each E-register is two bits wide, there are four possible states:

- The empty state (`_MPC_STC_EMPTY`) signals that the last E-register command using the corresponding E-register as source-and-destination register is still outstanding.
- The full-fault state (`_MPC_STC_FULL_F`) signals that the last E-register command using the corresponding E-register as source-and-destination register has completed with errors.
- The full state (`_MPC_STC_FULL`) signals that the last E-register command using the corresponding E-register as source-and-destination register has completed successfully.
- The full-send-reject state (`_MPC_STC_FULL_SR`) signals that the last send command using the corresponding E-register as source-and-destination register has been rejected.

The pending register located at `_MPC_MR_EREG_PENDING` provides a summary of the state for the individual E-registers in the two least-significant bits: The least significant bit of this register is set if there is at least one outstanding get command, cleared otherwise. The other bit of this register is set if there is at least one outstanding put command, cleared otherwise.

The E-register mechanism supports the following set of commands:

- The get command (`_MPC_EOP_ERS_GET`) returns the contents of a location in local or remote memory, i.e. performs a local or remote memory read.
- The put command (`_MPC_EOP_ERS_PUT`) updates the contents of a location in local or remote memory, i.e. performs a local or remote memory write.
- The swap command (`_MPC_EOP_MSWAP`) updates the contents of a location in local or remote memory and returns the original contents of the local or remote memory location.
- The conditional swap command (`_MPC_EOP_CSWAP`) updates the contents of a location in local or remote memory provided that the original contents meet the specified condition: In this case, the local or remote memory location is updated and the original contents are returned, similar to the swap command. Otherwise, the local or remote memory location remains unchanged and the original contents are returned.
- The fetch-and-increment command (`_MPC_EOP_GET_INC`) increments the contents of a location in local or remote memory and returns the original contents.
- The fetch-and-add command (`_MPC_EOP_GET_ADD`) adds the specified value to the contents of a location in local or remote memory and returns the original contents.
- The send command (`_MPC_EOP_SEND`) stores a message to a local or remote memory location. The specified memory location must contain a message

queue control word in order to ensure proper delivery of the message. The message queue control word defines a message queue of arbitrary size. This command is used to implement message-passing communication libraries, e.g. MPI [SOHL+98][GHLL+98], PVM [GBD+94].
- The state command (`_MPC_EOP_ERS_READ`) returns the contents of the specified state register.

The individual E-register commands described above can be combined with several qualifiers:

- The local qualifier (`_MPC_EOM_LOCAL`) can be combined with all commands except the state read if the corresponding command is guaranteed to access a local memory location.
- The vector qualifier (`_MPC_EOM_V8`) is implied by the send command and can be combined with the get and put commands to transfer blocks of eight E-registers with a single command: A get and put command combined with the vector qualifier uses eight consecutive source-and-destination registers to perform reads and writes to local or remote memory locations. This qualifier is useful for transferring large blocks of data, e.g. arrays.
- The longword qualifier(`_MPC_EOM_32BIT`) can be combined with all E-register commands except the send and state read commands. This qualifier indicates that the corresponding command accesses longword locations instead of the default quadword locations.
- The bypass qualifier (`_MPC_EOM_BYSTT`) can be combined with all E-register commands to bypass the segment translation table that is used to generate global addresses. As described in Section 6.1.2, the segment translation table is an integral part of the address generation mechanism, hence this qualifier is usually not used by the application programmer.
- The fixed qualifier (`_MPC_EOM_FIXOR`) can be combined with all E-register commands and ensures ordering between different E-register commands. All E-register commands using this qualifier are executed in the order in which they were issued.

It is possible to combine several of the above qualifiers with an E-register command, e.g. the get command in conjunction with the local and vector qualifiers.

An E-register command is issued by a write to a specific address. Both the address and the written data are used to specify the individual E-register command: The address depends on the command type as well as the used source-and-destination register. The data depends on the address of the local or remote memory location, the number of the local or remote processing element, as well as the used more-operands block.

The address is relative to the `MPC_EREG_BASE` base, the offset is determined as follows: Bits 13 to 21 are used to specify the command type and qualifiers, bits 0 to 12 are used to store the number of the source-and-destination

register. Note that the three least-significant bits are always zero, except for commands that use the vector qualifier.

The data uses the following format: Bits 56 to 63 specify the number of the more-operands block, bits 38 to 49 are used to store the number of the local or remote processing element, and the least-significant 38 bits are used to store the virtual address. Note that the number of the local or remote processing element is determined from a combination of the least-significant 50 bits, i.e. the number of the processing element and the virtual address, by the hardware centrifuge described in Section 6.1.2. This centrifuge is useful for distributing arrays across processors in a non-trivial way. The number of the processing element is taken from bits 38 to 49, if the default centrifuge mask is used.

B.2 E-Register Routines

The emulation library contains communication and synchronization routines that support a split-transaction protocol, i.e. separate issue and completion of local or remote memory requests. On the Cray T3E these routines are implemented by directly accessing the E-registers. As the individual routines use almost all E-register commands, they are well-suited to illustrate the aspects of E-register programming.

This section contains line-by-line descriptions of the individual communication and synchronization routines in the emulation library, i.e. the routines described in Section 2.2.3. Sections B.2.1, B.2.2, and B.2.3 cover the basic get and put routines. The conditional and unconditional swap routines are described in Sections B.2.4 and B.2.5, respectively. Sections B.2.6 and B.2.7 cover the fetch-and-add and fetch-and-increment routines. The routines described in Sections B.2.8 and B.2.9 provide information about the state of E-register operations. Note that these sections are intended to form a reference for the individual E-register commands and often contain similar material.

B.2.1 EMUereg_int_get()

The EMUereg_int_get() routine issues a get E-register command to a location in local/remote memory similar to the shmem_int_get() routine in the shmem library. In contrast to shmem_int_get(), this routine returns immediately after issuing the E-register command, i.e. without waiting for the local/remote memory transaction to complete.

```
void EMUereg_int_get(int ereg, const int *addr, int pe)
{
volatile long *Ecmd;
```

The EMUereg_int_get() routine expects three arguments:

ereg: the number of the source-and-destination register to use.
addr: the address of the local or remote memory location.
pe: the number of the local or remote processing element.

The `Ecmd` variable is used to issue the E-register command.

```
Ecmd  = (volatile long *)(_GET(_MPC_E_REG_SADE + ereg));
```

The address of the E-register command is initialized. Note that the address specifies the type of the E-register command as well as the E-register to use. The `_GET` macro is defined as

```
_GET(x) = (_MPC_E_REG_BASE | _MPC_EOP_GET | (x) << 3)
```

where `MPC_EREG_BASE` is the base address for all E-register operations. The `MPC_EOP_GET` macro sets the corresponding address bits for a get command. Since all E-register routines in the emulation library count the source-and-destination registers relative to the first source-and-destination register instead of the first more-operands block, the offset from the first source-and-destination register has to be added.

```
_write_memory_barrier();
```

The write memory barrier is an intrinsic for the WMB instruction and separates the initialization of the `Ecmd` variable above from the subsequent issue of the E-register command. This barrier is required as issuing both writes in reverse order results in a write to an unpredictable address.

```
*Ecmd = ((_MPC_E_REG_STRIDE1>>2) << _MPC_BS_EDATA_MOBE) |
        (                    (pe) << _MPC_BS_DFLTCENTPE) |
        (           (long)(addr)                        );
```

After the `Ecmd` variable has been initialized to the correct address, the get command is issued by a write to that address. Note that the address of this write specifies the E-register command as well as the source-and-destination register to use, while the written data specifies the corresponding more-operands block, the address of the memory location, as well as the processing element.

The EMUereg_int_get() routine uses the default more-operands block, which is read-only and provides a zero offset, a quadword stride, and a long-word addend for fetch-and-add operations. Since such a block consists of four E-registers, the number of the first E-register in the block is right-shifted by two bits to get the offset of the more-operands block. This offset is stored at the eight bits starting from bit `_MPC_BS_EDATA_MOBE`. The second line stores the number of the processing element in the twelve bits starting from bit `_MPC_BS_DFLTCENTPE`, since the default centrifuge mask is used. The address of the memory location is stored in the least-significant 38 bits before the data is written to the address specified by the `Ecmd` variable.

```
_memory_barrier();
}
```

This memory barrier is an intrinsic for the `mb` instruction and separates the issue of the E-register command above from the subsequent load of the source-and-destination register. This barrier is required as issuing both operations in reverse order causes the load of the E-register to return the wrong value. Since the E-register command is issued by a write and the command is completed by a read of the E-register, a write memory barrier is not sufficient.

B.2.2 EMUereg_int_load()

The EMUereg_int_load() routine returns the contents of the specified source-and-destination E-register that contains the results of a previous E-register command issued by one of the other E-register routines.

```
int EMUereg_int_load(int ereg)
{
volatile int *sade = ((volatile int*)_MPC_E_REG_BASE) +
                                     _MPC_E_REG_SADE;

sade += ereg;
```

The EMUereg_int_load() routine expects a single argument: the number of the source-and-destination E-register. The `sade` variable is initialized to the address of the first of these registers and is subsequently updated to point to the specified register. A simple addition is sufficient for this update, as the number of the register is stored in the least-significant part of the address.

```
memory_barrier();

return(*sade);
}
```

The contents of the specified source-and-destination E-register are retrieved by a load to the address stored in the `sade` variable and returned immediately. A write memory barrier is used to separate the update of the `sade` variable from the subsequent load of the E-register. Otherwise both operations could be executed in reverse order, causing the load of the E-register to return the contents of the first source-and-destination register instead of the specified E-register.

B.2.3 EMUereg_int_put()

The EMUereg_int_put() routine performs a remote put operation similar to the shmem_int_put() routine from the shmem library. In contrast to shmem_int_put(), the source-and-destination register used for the data transfer can be specified for the EMUereg_int_put() routine.

```
void EMUereg_int_put(int ereg, const int *addr,
                     const int value, int pe)
{
volatile int *sade = ((volatile int *)_MPC_E_REG_BASE) +
                                        _MPC_E_REG_SADE;
volatile long *Ecmd;
```

The EMUereg_int_put() routine expects four arguments:

ereg: the number of the source-and-destination register to use.
value: the value to store in the local or remote memory location.
addr: the address of the local or remote memory location.
pe: the number of the destination processing element.

The `sade` variable is initialized to the address of the first source-and-destination E-register and is subsequently updated to point to the specified E-register, while the `Ecmd` variable is used to issue the E-register command.

```
Ecmd  = (volatile long *)(_PUT(_MPC_E_REG_SADE + ereg));
sade += ereg;
```

The address of the E-register command is initialized. The `_PUT` macro is defined as

```
_PUT(x) = (_MPC_E_REG_BASE | _MPC_EOP_PUT | (x) << 3)
```

where `MPC_EREG_BASE` is the base address for all E-register operations. The `MPC_EOP_PUT` macro sets the corresponding address bits for a put command. Since the EMUereg routines count the source-and-destination registers relative to the first source-and-destination register instead of the first more-operands block, the offset from the first source-and-destination register has to be added. The `sade` variable is initialized to the address of the first source-and-destination E-register above and is subsequently updated to point to the specified E-register.

```
*sade = value;

_write_memory_barrier();
```

The value to store in the local/remote memory location is stored in the specified E-register by writing the value to the address given by the `sade` variable.

The write memory barrier is required to separate the initialization of the `Ecmd` and `sade` variables as well as the update of the E-register from the subsequent issue of the E-register put command. Otherwise, memory operations could be reordered, such that the issued E-register command references an unpredictable address or stores the unpredictable contents of the E-register to the destination.

```
*Ecmd = ((_MPC_E_REG_STRIDE1>>2) << _MPC_BS_EDATA_MOBE) |
        (                    (pe) << _MPC_BS_DFLTCENTPE) |
        (           (long)(addr)                       );
```

After the `Ecmd` variable has been initialized to the correct address, the put command is issued by a write to that address. The EMUereg_int_put() routine uses the default more-operands block, which is read-only and provides a zero offset, a quadword stride, as well as a longword addend for fetch-and-add operations. Since such a block consists of four E-registers, the number of the first E-register in the block is right-shifted by two bits to get the offset of the more-operands block. This offset is stored at the eight bits starting from bit `_MPC_BS_EDATA_MOBE`. The second line stores the number of the processing element in the twelve bits starting from bit `_MPC_BS_DFLTCENTPE`, since the default centrifuge mask is used. The address of the memory location is stored in the least-significant 38 bits, before the data is written to the address specified by the `Ecmd` variable.

```
_memory_barrier();
}
```

The memory barrier is required to separate the issue of the E-register command from subsequent accesses to the specified E-register.

B.2.4 EMUereg_int_cswap()

The EMUereg_int_cswap() routine issues a conditional swap, i.e. the contents of the local or remote memory location is only updated if the original contents are equal to the condition. The original contents of the local or remote memory location are returned in every case. The EMUereg_int_cswap() routine is similar to the shmem_int_cswap() routine from the shmem library. In contrast to shmem_int_cswap(), this routine returns immediately after issuing the E-register command, i.e. without waiting for the local or remote memory transaction to complete.

```
void EMUereg_int_cswap(int ereg, const int *addr,
                       int cond, int value, int pe)
{
volatile int *mobe = ((volatile int *)_MPC_E_REG_BASE) +
                                      _MPC_E_REG_SCR_MOBE;
volatile long *Ecmd;
```

The EMUereg_int_cswap() routine expects five arguments:

ereg: the number of the source-and-destination E-register to use.
addr: the address of the local or remote memory location.
cond: the value to compare with the content of the local or remote memory location.

value: the value to store in the local or remote memory location.
pe: the number of the destination processing element.

As the EMUereg_int_cswap routine does not use the default more-operands block, the `mobe` variable points to the first scratch more-operands block. The `Ecmd` variable is used to issue the E-register command.

```
Ecmd  = (volatile long *)(_CSWAP(_MPC_E_REG_SADE + ereg));
```

The address of the E-register command is initialized. The `_CSWAP` macro is defined as

```
_CSWAP(x) = (_MPC_E_REG_BASE | _MPC_EOP_CSWAP | (x) << 3)
```

where `MPC_EREG_BASE` is the base address for all E-register operations. The `MPC_EOP_CSWAP` macro sets the corresponding address bits for a conditional swap command. Since the EMUereg routines count the source-and-destination registers relative to the first source-and-destination register instead of the first more-operands block, the offset from the first source-and-destination register has to be added.

```
*(mobe + 0) = _MPC_E_DFLTCENT;
*(mobe + 1) = 0;
*(mobe + 2) = cond;
*(mobe + 3) = value;
```

The four E-registers in the scratch more-operands block are initialized for the subsequent conditional swap operation. As with all E-register routines, the default centrifuge mask is used together with a zero offset. The corresponding values are stored in the first and second E-register of the scratch block, respectively. The condition and value arguments are stored in the third and fourth E-register. Note that the conditional swap operation uses a non-standard allocation for the last two E-registers: The condition is placed in the stride field, while the value is placed in the addend field.

```
_write_memory_barrier();
```

The write memory barrier is required to separate the initialization of the `Ecmd` and `mobe` variables as well as the update of the more-operands block of E-registers from the subsequent issue of the E-register conditional swap command. Otherwise, memory operations could be reordered, such that the issued E-register command references an unpredictable address, uses an unpredictable condition, or stores an unpredictable value in the local or remote memory location.

```
*Ecmd = ((_MPC_E_REG_SCR_MOBE>>2) << _MPC_BS_EDATA_MOBE) |
        (                    (pe) << _MPC_BS_DFLTCENTPE) |
        (          (long)(target)                      );
```

After the `Ecmd` variable has been initialized to the correct address, the conditional swap command is issued by a write to that address. The EMUereg_int_cswap() routine uses the first scratch more-operands block of E-registers, which was initialized above. Since such a block consists of four E-registers, the number of the first E-register in the block is right-shifted by two bits to get the offset of the block. This offset is stored at the eight bits starting from bit `_MPC_BS_EDATA_MOBE`. The second line stores the number of the processing element in the twelve bits starting from bit `_MPC_BS_DFLTCENTPE`, since the default centrifuge mask is used. The address of the memory location is stored in the least-significant 38 bits, before the data is written to the address specified by the `Ecmd` variable.

```
_memory_barrier();
}
```

The memory barrier is required to separate the issue of the E-register command from subsequent accesses to the specified E-register.

B.2.5 EMUereg_int_mswap()

The EMUereg_int_mswap() routine issues a memory swap, i.e. the contents of the local or remote memory location are updated and the original contents of the local or remote memory location are returned. The EMUereg_int_mswap() routine is similar to the shmem_int_mswap() routine from the shmem library. In contrast to shmem_int_mswap(), this routine returns immediately after issuing the E-register command, i.e. without waiting for the local or remote memory transaction to complete.

```
void EMUereg_int_mswap(int ereg, const int *addr,
                       int value, int pe)
{
volatile int *mobe = ((volatile int *)_MPC_E_REG_BASE) +
                                      _MPC_E_REG_SCR_MOBE;
volatile long *Ecmd;
```

The EMUereg_int_mswap() routine expects four arguments:

ereg: the number of the source-and-destination E-register to use.
addr: the address of the local or remote memory location.
value: the value to store in the local or remote memory location.
pe: the number of the destination processing element.

As the EMUereg_int_mswap() routine does not use the default more-operands block, the `mobe` variable points to the first scratch more-operands block. The `Ecmd` variable is used to issue the E-register command.

```
Ecmd  = (volatile long *)(_MSWAP(_MPC_E_REG_SADE + ereg));
```

The address of the E-register command is initialized. The `_MSWAP` macro is defined as

```
_MSWAP(x) = (_MPC_E_REG_BASE | _MPC_EOP_MSWAP | (x) << 3)
```

where `MPC_EREG_BASE` is the base address for all E-register operations. The `MPC_EOP_CSWAP` macro sets the corresponding address bits for a masked swap command. Since the E-register routines in the emulation library count the source-and-destination registers relative to the first source-and-destination register instead of the first more-operands block, the offset from the first source-and-destination register has to be added.

```
*(mobe + 0) = _MPC_E_DFLTCENT;
*(mobe + 1) = 0;
*(mobe + 2) = -1;
*(mobe + 3) = value;
```

The four E-registers in the scratch more-operands block are initialized for the subsequent conditional swap operation. As with all E-register routines, the default centrifuge mask is used together with a zero offset. The corresponding values are stored in the first and second E-register of the scratch block, respectively. The value is stored in the fourth E-register, while the third E-register is initialized to all ones to ensure proper operation of the unconditional swap command. Note that the unconditional swap operation uses a non-standard allocation for the last two E-registers: The stride field is used as a bit mask, while the value is placed in the addend field.

```
_write_memory_barrier();
```

The write memory barrier is required to separate the initialization of the `Ecmd` and `mobe` variables as well as the update of the more-operands block of E-registers from the subsequent issue of the E-register unconditional swap command. Otherwise, memory operations could be reordered, such that the issued E-register command references an unpredictable address or stores an unpredictable value in the local or remote memory location.

```
*Ecmd = ((_MPC_E_REG_SCR_MOBE>>2) << _MPC_BS_EDATA_MOBE) |
        (                     (pe) << _MPC_BS_DFLTCENTPE) |
        (          (long)(target)                       );
```

After the `Ecmd` variable has been initialized to the correct address, the masked swap command is issued by a write to that address. The EMUereg_int_mswap() routine uses the first scratch more-operands block of E-registers, which was initialized above. Since such a block consists of four E-registers, the number of the first E-register in the block is right-shifted by two bits to get the offset of the block. This offset is stored at the eight bits starting from bit `_MPC_BS_EDATA_MOBE`. The second line stores the number of the processing element in the twelve bits starting from bit `_MPC_BS_DFLTCENTPE`,

since the default centrifuge mask is used. The address of the memory location is stored in the least-significant 38 bits, before the data is written to the address specified by the `Ecmd` variable.

```
_memory_barrier();
}
```

The memory barrier is required to separate the issue of the E-register command from subsequent accesses to the specified E-register.

B.2.6 EMUereg_int_finc()

The EMUereg_int_finc() routine issues a fetch-and-increment command, i.e. returns the original contents of the local or remote location as well as incrementing the contents of the local or remote memory location. The EMUereg_int_finc() routine is similar to the shmem_int_finc() routine from the shmem library. In contrast to shmem_int_finc(), this routine returns immediately after issuing the E-register command, i.e. without waiting for the local or remote memory transaction to complete.

```
void EMUereg_int_finc(int ereg, const int *addr, int pe)
{
volatile long *Ecmd;
```

The EMUereg_int_finc() routine expects three arguments:

ereg: the number of the source-and-destination E-register to use.
addr: the address of the local or remote memory location.
pe: the number of the destination processing element.

The `Ecmd` variable is used to issue the E-register command.

```
Ecmd  = (volatile long *)(_GET_INC(_MPC_E_REG_SADE + ereg));
```

The address of the E-register command is initialized. Note that the address specifies the type of the E-register command as well as the E-register to use. The `_GET_INC` macro is defined as

```
_GET_INC(x) = (_MPC_E_REG_BASE  |
               _MPC_EOP_GET_INC | (x) << 3)
```

where `MPC_EREG_BASE` is the base address for all E-register operations. The `MPC_EOP_GET_INC` macro sets the corresponding address bits for a fetch and increment command. Since the E-register routines in the emulation library count the source-and-destination registers relative to the first source-and-destination register instead of the first more-operands block, the offset from the first source-and-destination register has to be added.

```
_write_memory_barrier();
```

The write memory barrier separates the initialization of the `Ecmd` variable above from the subsequent issue of the E-register command. This barrier is required as both writes may be issued in reverse order otherwise, causing a write to an unpredictable address.

```
*Ecmd = ((_MPC_E_REG_STRIDE1>>2) << _MPC_BS_EDATA_MOBE) |
        (                   (pe) << _MPC_BS_DFLTCENTPE) |
        (         (long)(target)                      );
```

After the `Ecmd` variable has been initialized to the correct address, the fetch-and-increment command is issued by a write to that address. The EMUereg_int_finc() routine uses the default more-operands block of E-registers, which is read-only and provides a zero offset, a quadword stride, as well as a longword addend for fetch-and-add operations. Since such a block consists of four E-registers, the number of the first E-register in the block is right-shifted by two bits to get the offset of the block. This offset is stored at the eight bits starting from bit `_MPC_BS_EDATA_MOBE`. The second line stores the number of the processing element in the twelve bits starting from bit `_MPC_BS_DFLTCENTPE`, since the default centrifuge mask is used.The address of the memory location is stored in the least-significant 38 bits, before the data is written to the address specified by the `Ecmd` variable.

```
_memory_barrier();
}
```

This memory barrier separates the issue of the E-register command above from the subsequent load of the E-register. This barrier is required as both operations could be issued in reverse order otherwise, causing the load of the E-register to return the wrong value. Since the E-register command is issued by a write and the command is completed by a read of the E-register, a write memory barrier is not sufficient.

B.2.7 EMUereg_int_fadd()

The EMUereg_int_fadd() routine issues a fetch-and-add operation, i.e. returns the contents of the local or remote memory location while updating the contents of the local or remote memory location by adding the specified value. The EMUereg_int_fadd() routine is similar to the shmem_int_fadd() routine from the shmem library. In contrast to shmem_int_fadd(), this routine returns immediately after issuing the E-register command, i.e. without waiting for the local or remote memory transaction to complete.

```
void EMUereg_int_fadd(int ereg, const int *addr,
                      int value, int pe)
{
volatile int *mobe = ((volatile int *)_MPC_E_REG_BASE) +
```

```
                                                _MPC_E_REG_SCR_MOBE;
volatile long *Ecmd;
```

The EMUereg_int_fadd() routine expects four arguments:

ereg: the number of the source-and-destination register to use.
addr: the address of the local or remote memory location.
value: the value to add to the local or remote memory location.
pe: the number of the destination processing element.

The EMUereg_int_fadd() routine does not use the default more-operands block, hence the `mobe` variable points to the first scratch more-operands block. The `Ecmd` variable is used to issue the E-register command.

```
Ecmd  = (volatile long *)(_GET_ADD(_MPC_E_REG_SADE + ereg));
```

The address of the E-register command is initialized. The `_GET_ADD` macro is defined as

```
_GET_ADD(x) = (_MPC_E_REG_BASE  |
               _MPC_EOP_GET_ADD | (x) << 3)
```

where `MPC_EREG_BASE` is the base address for all E-register operations. The `MPC_EOP_GET_ADD` macro sets the corresponding address bits for a fetch-and-add command. Since the EMUereg routines count the source-and-destination registers relative to the first source-and-destination register instead of the first more-operands block, the offset from the first source-and-destination register has to be added.

```
*(mobe + 0) = _MPC_E_DFLTCENT;
*(mobe + 1) = 0;
*(mobe + 2) = -1;
*(mobe + 3) = value;
```

The four E-registers in the scratch more-operands block are initialized for the subsequent fetch-and-add operation. As with all E-register routines in the emulation library, the default centrifuge mask is used together with a zero offset. The corresponding values are stored in the first and second E-register of the scratch block, respectively. The value is stored in the fourth E-register, while the third E-register is initialized to all ones to ensure proper operation of the fetch-and-add command. Note that the fetch-and-add operation uses the same non-standard allocation for the last two E-registers as the unconditional swap operation.

```
_write_memory_barrier();
```

The write memory barrier is required to separate the initialization of the `Ecmd` and `mobe` variables as well as the update of the more-operands block of E-registers from the subsequent issue of the E-register fetch-and-add

command. Otherwise memory operations could be reordered, such that the issued E-register command references an unpredictable address or stores an unpredictable value in the local/remote memory location.

```
*Ecmd = ((_MPC_E_REG_SCR_MOBE>>2) << _MPC_BS_EDATA_MOBE) |
        (                    (pe) << _MPC_BS_DFLTCENTPE) |
        (          (long)(target)                      );
```

After the `Ecmd` variable has been initialized to the correct address, the fetch-and-add command is issued by a write to that address. The routine uses the first scratch more-operands block of E-registers, which was initialized above. Since such a block consists of four E-registers, the number of the first E-register in the block is right-shifted by two bits to get the offset of the block. This offset is stored at the eight bits starting at bit `_MPC_BS_EDATA_MOBE`. The second line stores the number of the processing element in the twelve bits starting at bit `_MPC_BS_DFLTCENTPE`, since the default centrifuge mask is used. The address of the memory location is stored in the least-significant 38 bits, before the data is written to the address specified by the `Ecmd` variable.

```
_memory_barrier();
}
```

The memory barrier is required to separate the issue of the E-register command from subsequent accesses to the specified E-register.

B.2.8 EMUereg_pending()

The EMUereg_pending() routine reads the `_MPC_MR_EREG_PENDING` register. Most of the bits in this register are unused, the information is stored in the two least-significant bits: The least-significant bit is set if there are any outstanding get-type operations, the other is set if there are any outstanding put-type operations. This register can therefore be used to check for the completion of all outstanding E-register commands, without the need to check the status of each individual E-register. The shmem library does not contain a similar routine, since all outstanding operations must be completed before any of the shmem routines return.

```
long EMUereg_pending(void )
{
return(*((volatile long *)_MPC_MR_EREG_PENDING));
}
```

The EMUereg_pending() routine expects no arguments and immediately returns the contents of the `_MPC_MR_EREG_PENDING` register. Note that this register is addressed relative to the `_MPC_MR_BASE`, instead of the `_MPC_E_REG_BASE` used for all other E-register operations.

B.2.9 EMUereg_state()

The EMUereg_state() routine is similar to the EMUereg_pending() routine described above, but provides more detailed information: Instead of checking if there are any outstanding get or put operations, the EMUereg_state() routine checks whether there is an outstanding E-register command for the specified E-register. This is accomplished by reading the corresponding state register and extracting the state for the specified E-register. The shmem library does not contain a comparable routine, since all outstanding operations must be completed before any of the shmem routines return.

```
long EMUereg_state(int ereg)
{
volatile long *Ecmd;
long state;
```

The EMUereg_state() routine expects a single argument, i.e. the number of the E-register to be checked. The `Ecmd` variable is used to issue the E-register command, while the `state` variable is used to store the state of the specified E-register.

```
Ecmd = (volatile long *)(_ERS(ereg >> 5));
```

The `Ecmd` variable is initialized to the address of the state register that contains the state of the specified E-register. The `_ERS` macro is defined as

```
_ERS(x) = (_MPC_EOP_ERS_READ |
           _MPC_E_REG_BASE   | (sade << 3))
```

which is based on experimentation, as this macro is not defined by the header files mentioned above. Note that the macro expects the number of the state register instead of the number of the E-register to be checked. However, the mapping between state registers and E-registers is straight-forward: Every state register contains the state for 32 E-registers, hence the number of the E-register is right-shifted by 5 bits to obtain the number of the corresponding state register.

```
_write_memory_barrier();
```

The write memory barrier is required to separate the initialization of the `Ecmd` variable from the subsequent issue of the E-register state read command. Otherwise memory operations could be reordered, such that the issued E-register command references an unpredictable address.

```
state = ((*Ecmd) >> (ereg % 32)) & (~0x3);

return(state);
}
```

The E-register state read is issued by a read to the address stored in the `Ecmd` variable, which returns the contents of the specified state register. The state regarding the specified E-register is isolated by right-shifting the state to the two least-significant bit positions and subsequently clearing all other bit positions. The two bits of state for each E-register encode the four different states that have been described in Section B.1.

B.3 Programming Guidelines

The E-registers are the only way to access remote memory on the Cray T3E. Therefore all communication packages that are available on the T3E use the E-registers. Apart from remote memory accesses, the E-registers can also be used for single-processor optimizations: Since the get and put commands operate in the uncached memory region, E-registers can be used to transfer large amounts of memory without destroying the contents of the cache. Several other useful optimizations using the E-registers are described in the Cray T3E optimization guides [ABH97][Cra97b]. The compiler may generate code that uses the E-registers, e.g. if the `cache_bypass` directive is used. In conclusion, the E-registers are not only used by communication libraries, but might also be used by other libraries or the compiler itself. Therefore accesses to the E-register from several sources must be separated to ensure proper operation.

In order to avoid conflicting accesses to the E-registers, the accesses from different libraries have to be separated in time. However, most libraries restrict accesses to the E-registers within subroutine calls, i.e. all E-register operations are completed prior to subroutine return. The only exception to this rule is the benchlib as well as the emulation library. Since the benchlib is not available to the general public, this library is not considered further.

In order to separate accesses to the E-registers in time, it is sufficient to complete all accesses that were initiated by the emulation library, before calling a subroutine from one of the other libraries. This resolves the conflicts between the emulation library and other libraries, but the compiler itself may still generate code that can cause conflicting accesses to the E-registers.

At present, the compiler only generates code that uses the E-registers if explicitly told to do so, e.g. via the `cache_bypass` directive. Procedures that use one of these directives should therefore be treated in the same way as library functions that contain accesses to the E-registers. However, future versions of the compiler might generate conflicting code automatically, i.e. without being prompted by directives.

The source code of the routines in the emulation library contains directives that forbid the compiler to use those E-registers, that are marked as used. The `mobe(`n`)` and `sade(`n`)` directives mark the first n more-operand-blocks and source-and-destination E-registers as used, respectively. Therefore following

the rules outlined above will ensure that conflicting accesses to the E-registers are avoided.

Apart from the conflicts mentioned above, there is a serious flaw in early versions of the Cray T3E that may be triggered by using the E-registers directly. As the consequences of this flaw can be fatal, i.e. it is possible to crash the whole machine, extreme caution is required when using the emulation library on these versions of the Cray T3E. The following paragraphs contains a short summary about the flaw as well as possible workarounds. A white-paper from Cray provides more detailed information [Cra96].

The flaw affects the Cray T3E-600, i.e. the initial version of the T3E using a processor clock frequency of 300 MHz. The problem has been solved in the T3E-900, i.e. the version of the T3E using a processor clock frequency of 450 MHz, and later models. The problem manifests itself if an E-register command references a memory location that is simultaneously cached in one of the stream buffers. Note that the offending E-register command may originate from the local processor element or any of the remote processing elements. If this situation occurs, the system logic on the affected processing element looses track about cache coherence. This can cause the crash of the affected processing node, the local manager node, and ultimately the whole system via the global manager node.

But the problem is even worse: It is not sufficient to avoid simultaneous cached and uncached accesses to the same memory locations via the stream buffers and the E-registers, respectively: Since the stream buffers are used to prefetch data, they may contain memory locations that are ahead of previous accessed memory locations. Therefore simultaneous cached and uncached references to nearby memory locations are sufficient to trigger the flaw. More specifically, uncached references should access memory locations that are below or at least 192 bytes above locations that are simultaneously accessed by cached references. This problem can be addressed in three different ways, i.e. by disabling the stream buffers or by separating cached and uncached references in space and time, respectively.

The first solution does not require any program changes, but may severely impact the performance of the system, especially on vectorizable codes. However, most installations that still use one of the T3E-600 models use this solution. A less radical approach is to enable stream buffers by default, but ensure that all programs disable the stream buffers before entering a program section, that may contain cached and uncached references to similar memory locations. This can be accomplished by using the `set_d_stream()` and `quiet_d_stream()` routines.

The second solution is to separate conflicting accesses in space. This is accomplished by identifying all arrays that may be in use while uncached references to other arrays occur, and provide sufficient padding between these arrays. The drawback of this solution is the amount of work required to identify and separate such arrays.

The third solution is to separate conflicting accesses in time. This is accomplished by placing synchronization barriers between sections that may contain cached and uncached references to similar memory locations. Alternatively, conflicting data structures can be protected by synchronization locks. The drawback of this solution is the impact on performance as well as the work required to modify the program.

In conclusion, the use of the emulation library on Cray T3E-600 models with stream buffers enabled by default is strongly discouraged and no liability of any form is assumed if the emulation library is used under these conditions.

Literature

[ABB+86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Technical Conference*, pages 93–112, July 1986.

[ABBJ64] G. M. Amdahl, G. A. Blauuw, and F. P. Brooks Jr. Architecture of the IBM System/360. *IBM Journal of Research and Development*, 8(2):87–101, April 1964.

[ABC+95] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22th International Symposium on Computer Architecture (ISCA)*, pages 2–13, June 1995.

[ABD+97] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous Profiling: Where Have All the Cycles Gone. Technical Report SRC 1997-016a, Digital Systems Research Center, September 1997.

[ABH97] Ed Anderson, Jeff Brooks, and Tom Hewitt. *The Benchmarker's Guide to Single-Processor Optimization for Cray T3E Systems.* Cray Research Inc., 1997.

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[ACC+90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. In *Proceedings of Supercomputing*, pages 1–6, November 1990.

[ACK+95] Remzi H. Arpaci, David E. Culler, Arvind Krishnamurthy, Steve G. Steinberg, and Katherine Yelick. Empirical Evaluation of the Cray-T3D: A Compiler Perspective. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*, pages 320–331, June 1995.

[ADK+92] F. Abolhassan, R. Drefenstedt, J. Keller, W. J. Paul, and D. Scheerer. On the Physical Design of PRAMs. *Computer Journal*, 36(8):756–762, December 1992.

[Aga92] Anant Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[AKK+95] Robert Alverson, Simon Kahan, Richard Korry, Cathy McCann, and Burton Smith. Scheduling on the Tera MTA. In *Proceedings of the IPPS Workshop on Job Scheduling Strategies for Parallel Processing*, pages 19–44, April 1995.

[Alf94] Robert A. Alfieri. An Efficient Kernel-Based Implementation of POSIX Threads. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 59–72, June 1994.

[ALKK90] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*, pages 104–114, June 1990.

[ALL89] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[AMBN98] Ernest Artiaga, Xavier Martorell, Yolanda Becerra, and Nacho Navarro. Experiences on the Implementing PARMACS Macros to Run the SPLASH-2 Suite on Multiprocessors. In *Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing*, pages –, January 1998.

[ANMB97] Ernest Artiaga, Nacho Navarro, Xavier Martorell, and Yolanda Becerra. Implementing PARMACS Macros for Shared Memory Multiprocessor Environments. Technical Report UPC-DAC-1997-07, Polytechnic University of Catalunya, Department of Computer Architecture, January 1997.

[AXP96a] Digital Equipment Corp. *Alpha 21064 and 21064A Microprocessors Hardware Reference Manual*, June 1996.

[AXP96b] Digital Equipment Corp. *Alpha 21066 and 21066A Microprocessors Hardware Reference Manual*, January 1996.

[AXP96c] Digital Equipment Corp. *Alpha 21164 Microprocessor Hardware Reference Manual*, July 1996.

[AXP97] Digital Equipment Corp. *Alpha 21164PC Microprocessor Hardware Reference Manual (PCA56)*, September 1997.

[AXP98a] Compaq Computer Corp. *Alpha 21164A Microprocessor Hardware Reference Manual*, December 1998.

[AXP98b] Compaq Computer Corp. *Alpha 21164PC Microprocessor Hardware Reference Manual (PCA57)*, December 1998.

[AXP99] Compaq Computer Corp. *Tsunami/Typhoon 21272 Chipset Hardware Reference Manual*, October 1999.

[AXP00a] Compaq Computer Corp. *Alpha 21264/EV6 Microprocessor Hardware Reference Manual*, September 2000.

[AXP00b] Compaq Computer Corp. *Alpha 21264/EV67 Microprocessor Hardware Reference Manual*, September 2000.

[AXP00c] Compaq Computer Corp. *Alpha 21264/EV68 Microprocessor Hardware Reference Manual*, December 2000.

[Bai90] David H. Bailey. FFTs in External or Hierarchical Memory. *The Journal of Supercomputing*, 4(1):23–35, March 1990.

[Ban99] Peter Bannon. Alpha 21364: A Scalable Single-chip SMP. In *Proceedings of Microprocessor Forum*, pages –, October 1999.

[BBD+87] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfeld, Ewing Lusk, Ross Overbeck, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, Inc., 1987.

[BBF+97] P. Bach, M. Braun, A. Formella, J. Friedrich, Th. Grn, and C. Lichtenau. Building the 4 Processor SB-PRAM Prototype. In *Proceedings of the Hawaii 30th International Symposium on System Science (HICSS)*, pages 14–23, January 1997.

[BBLS91] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report RNR-91-02, NASA Ames Research Center, January 1991.

[BC91a] Jean-Loup Baer and Tien-Fu Chen. An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty. In *Proceedings of Supercomputing 91*, pages 176–186, November 1991.

[BC91b] Dileep P. Bhandarkar and Douglas W. Clark. Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization. In *Proceedings Fourth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 310–319, April 1991.

[BCD+92] David S. Blickstein, Peter W. Craig, Caroline S. Davidson, R. Neil Faiman, Kent D. Glossop, Richard B. Grove, Steven O. Hobbs, and William B. Noyce. The GEM Optimizing Compiler System. *Digital Technical Journal*, 4(4):121–136, Fall 1992.

[BEH91] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrated Register Allocation and Instruction Scheduling for RISCs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 122–131, April 1991.

[BEKK00] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded PowerPC processor for commercial servers. *IBM Journal of Research and Development*, 44(6):885–897, November 2000.

[BG95] Thomas J. Bergin and Richard G. Gibson, editors. *The History of Programming Languages-II.* ACM Press, 1995.

[BH86] Josh Barnes and Piet Hut. A hierarchical $o(n \log n)$ force-calculation algorithm. *Nature*, 324(4):446–449, December 1986.

[BHS+95] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.

[BJK+96] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[BL96] Ricardo Bianchini and Beng-Hong Lim. Evaluating the Performance of Multithreading and Prefetching in Multiprocessors. *Journal of Parallel and Distributed Computing*, 37(1):83–97, August 1996.

[Bli96] Bruce Blinn. *Portable Shell Programming.* Prentice-Hall, 1996.

[BLL88] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A System for Object-oriented Parallel Programming. *Software Practice & Experience*, 18(8):713–732, August 1988.

[BLM+91] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Symposium on Parallel Algorithms and Architectures*, pages 3–16, July 1991.

[BM98] Rudolf Berrendorf and Bernd Mohr. PCL - The Performance Counter Library. Technical Report IB-9816, Zentrum für Angewandte Mathematik, Forschungszentrum Jülich, October 1998.

[BOW+90] Dileep P. Bhandarkar, D. Orbits, R. Witek, W. Cardoza, and D. Cutler. High Performance Issue Oriented Architecture. In *Proceedings of Compcon Spring 1997*, pages 153–160, February 1990.

[BR91] David Bernstein and Michael Rodeh. Global Instruction Scheduling for Superscalar Machines. In *Proceedings ACM Conference on Programming Language Design and Implementations (PLDI)*, pages 241–255, June 1991.

[BR92] Bob Boothe and Abhiram Ranade. Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*, pages 214–223, May 1992.

[Bur01] Tom Burd. CPU Info Center: General Processor Information . http://bwrc.eecs.berkeley.edu/CIC/summary/local/summary.pdf, 2001.

[But97] David R. Butenhof. *Programming with POSIX Threads*. Addison Wesley Longman, 1997.

[CAC+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register Allocation via Coloring. *Computer Languages*, 6(1):47–57, January 1981.

[Cat91] Ben J. Catanzaro, editor. *The SPARC Technical Papers*. Springer Verlag, 1991.

[CB94] Tien-Fu Chen and Jean-Loup Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, pages 223–232, April 1994.

[CD88] Eric C. Cooper and Richard P. Draves. C Threads. Technical Report CMU-CS-88-154, Carnegie Mellon University, February 1988.

[CDD+89] Robert Conrad, Richard Devlin, Daniel Dobberpuhl, Bruce Gieseke, Richard Heye, Gregory Hoeppner, John Kowaleski, Maureen Ladd, James Montanaro, Steve Morris, Rebecca Stamm, Henry Tumblin, and Richard Witek. A 50 MIPS (Peak) 32b/64b Microprocessor. In *Digest International Solid-State Circuits Conference (ISSCC)*, pages 76–77, February 1989.

[CH84] Frederick Chow and John Hennessy. Register Allocation by Priority-based Coloring. In *Proceedings of the ACM Symposium on Compiler Construction*, pages 222–232, June 1984.

[Cha82] G J. Chaitin. Register Allocation & Spilling via Graph Coloring. In *Proceedings of the ACM Symposium on Compiler Construction*, pages 98–105, June 1982.

[CL82] Douglas W. Clark and Henry M. Levy. Measurement and Analysis of Instruction Use in the VAX-11/780. In *Proceedings Ninth Symposium on Computer Architecture*, pages 9–17, April 1982.

[CL90] Fred C. Chow and Hennessy. John L. The Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[Com98] Alpha Architecture Commitee. *Alpha Architecture Reference Manual*. Digital Press, third edition, 1998.

[Cor99] Workstation Marketing Compaq Computer Corporation. *Compaq Professional Workstation XP1000 Key Technologies White Paper*. Compaq Computer Corporation, January 1999.

[Cor00] High Performance Technical Computing Group Compaq Computer Corporation. *AlphaServerSC: Scalable Supercomputing*. Compaq Computer Corporation, July 2000.

[Cra96] Cray Research Inc. *Cray T3E Programming with Coherent Memory Streams*, 1996.

[Cra97a] hwdefs_t3e.h header file. Cray Research Inc., 1997.

[Cra97b] Cray Research Inc. *Cray T3E C and C++ Optimization Guide*, 1997.

[Cra97c] Cray Research Inc. *System Architecture Kit (Cray T3E System)*, 1997.

[Cra98a] mpphw_t3e.h header file. Cray Research Inc., 1998.

[Cra98b] intro_shmem(3) man page. Cray Research Inc., 1998.

[Cra98c] shmem.h header file. Cray Research Inc., 1998.

[CS99] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[CSS+91] David E. Culler, Anurag Sah, Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 164–175, April 1991.

[Dal90] William J. Dally. Virtual-Channel Flow Control. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*, pages 60–68, May 1990.

[DFK+92] William J. Dally, J. A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Gergory A. Fyler. The Message-Driven Processor. *IEEE Micro*, 12(2):23–38, April 1992.

[DH95] J. J. Dongarra and T. Hey. The ParkBench Benchmark Collection. *Supercomputer*, 11(2–3):94–114, June 1995.

[DHW+97] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO)*, pages 292–302, December 1997.

[Die99] Keith Diefendorff. Compaq Chooses SMT for Alpha. *Microprocessor Report*, 13(16):1–11, December 1999.

[DM91] Peter A. Darnell and Philip E. Margolis. *C: A Software Engineering Approach.* Springer Verlag, 2nd. edition, 1991.

[DO95] Mikhail N. Dorojevets and Vojin G. Oklobdzija. Multithreaded Decoupled Architecture. *International Journal of High Speed Computing*, 7(3):465–480, September 1995.

[Don90] J. J. Dongarra. The LINPACK benchmark: an explanation. In *Evaluating Supercomputers: Strategies for Exploiting, Evaluating and Benchmarking Computers with Advanced Architecture*, Unicom Applied Information Technology Reports, pages 1–21. Chapmann & Hall, 1990.

[DR97] Dale Dougherty and Arnold Robbins, editors. *sed & awk.* O'Reilly and Associates, Inc., 2nd. edition, 1997.

[DS01] Jack J. Dongarra and Mathematical Sciences Section. Performance of Various Computers Using Standard Linear Equations software. Technical Report CS-89-85, University of Tennessee, July 2001.

[DW92] Mikhail N. Dorozhevets and Peter Wolcott. The El'brus-3 and MARS-M: Recent Advances in Russian High-Performance Computing. *The Journal of Supercomputing*, 6(1):5–48, March 1992.

[DWA+92a] Daniel W. Dobberpuhl, Richard W. Witek, Randy Allmon, Robert Anglin, David Bertucci, Sharon Britton, Linda Chao, Robert A. Conrad, Daniel E. Dever, Bruce Gieseke, Soha M. N. Hassoun, Gregory W. Hoeppner, Kathryn Kuchler, Maureen Ladd, Burton M. Leary, Liam Madden, Edward J. McLellan, Derrick R. Meyer, James Montanaro, Donald A. Priore, Vidya Rajagopalam, Sridhar Samudrale, and Sribalan Santhanam. A 200-MHz 64-b Dual-Issue CMOS Microprocessor. *IEEE Journal of Solid-State Circuits*, 27(11):1555–1567, November 1992.

[DWA+92b] Daniel W. Dobberpuhl, Richard W. Witek, Randy Allmon, Robert Anglin, David Bertucci, Sharon Britton, Linda Chao, Robert A. Conrad, Daniel E. Dever, Bruce Gieseke, Soha M. N. Hassoun, Gregory W. Hoeppner, Kathryn Kuchler, Maureen Ladd, Burton M. Leary, Liam Madden, Edward J. McLellan, Derrick R. Meyer, James Montanaro, Donald A. Priore, Vidya Rajagopalam, Sridhar Samudrale, and Sribalan Santhanam. A 200-MHz 64-bit Dual-Issue CMOS Microprocessor. *Digital Technical Journal*, 4(4):35–50, Fall 1992.

[DZU98a] Bernd Dreier, Markus Zahn, and Theo Ungerer. Parallel and Distributed Programming with Pthreads and Rthreads. In *Proceedings of the 3rd International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 34–40, March 1998.

[DZU98b] Bernd Dreier, Markus Zahn, and Theo Ungerer. The Rthreads Distributed Shared Memory System. In *Proceedings of the 3rd International Conference on Massively Parallel Computing Systems*, pages –, April 1998.

[EEL+97] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca Stamm, and Dean M. Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. *IEEE Micro*, 17(5):12–19, September 1997.

[EGK+94] Kemal Ebciogli, Randy Groves, Ki-Chang Kim, Gabriel Silberman, and Isaac Ziv. VLIW Compilation Techniques in a Superscalar Environment. In *Proceedings of the 1994 Symposium on Programming Languages Design and Implementation*, pages 36–46, July 1994.

[ERB+95] John H. Edmondson, Paul I. Rubinfeld, Peter J. Bannon, Bradley J. Benschneider, Debra Bernstein, Ruben W. Castelino, Elizabeth M. Cooper, Daniel E. Dever, Dale R. Donchin, Timothy C. Fischer, Anil K. Jain, Shekhar Mehta, Jeanne E. Meyer, Ronald P. Preston, Vidya Rajagopalan, Chandrasekhara Somanathan, Scott A. Taylor, and Gilbert M. Wolrich. Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-Issue CMOS RISC Microprocessor. *Digital Technical Journal*, 7(1):119–135, Winter 1995.

[ERPR95] John H. Edmondson, Paul Rubinfeld, Ronal Preston, and Vidya Rajagopalan. Superscalar Instruction Execution in the 21164 Alpha Microprocessor. *IEEE Micro*, 15(2):33–43, April 1995.

[EV97] Roger Espasa and Mateo Valero. Exploiting Instruction- and Data-Level Parallelism. *IEEE Micro*, 17(5):20–27, September 1997.

[EZ93] Derek L. Eager and John Zahorjan. Chores: Enhanced Run-Time Support for Shared Memory Parallel Computing. *ACM Transactions on Computer Systems*, 11(1):1–32, February 1993.

[Feo88] John T. Feo. An analysis of the computational and parallel complexity of the Livermore Loops. *Parallel Computing*, 7(2):163–185, June 1988.

[FF92] Joseph A. Fisher and Stefan M. Freudenberger. Predicting Conditional Branch Directions From Previous Runs of a Program. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 85–95, October 1992.

[Fis81] Joseph A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[FKD95] Marco Fillo, Stephen W. Keckler, and William J. Dally. The M-Machine Multicomputer. In *Proceedings of the 28th International Symposium on Microarchitecture (MICRO)*, pages 146–156, November 1995.

[FLA94] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementations (OSDI)*, pages 201–213, November 1994.

[Fly95] Michael J. Flynn. *Computer Architecture : Pipelined and Parallel Processor Design*. Jones and Bartlett Publishers, 1995.

[FM92] Edward W. Felten and Dylan McNamee. Improving the Performance of Message-Passing Applications by Multithreading. In *Proceedings of the Scalable High Performance Computing Conference*, pages 84–89, April 1992.

[Fre74] R. A. Freiburghouse. Register Allocation via Usage Counts. *Communications of the ACM*, 17(11):638–642, November 1974.

[GB96] Manu Gulati and Nader Bagherzadeh. Performance Study of a Multithreaded Superscalar Microprocessor. In *Proceedings of the 2nd International*

Symposium on High-Performance Computer Architecture (HPCA), pages 291–301, February 1996.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Bob Mancheck, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[GHG+91] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, pages 254–263, May 1991.

[GHLL+98] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Spahir, and Marc Snir. *MPI - The Complete Reference*, volume 2. MIT Press, 1998.

[GJ79] Michael R. Garey and David S. Johnson, editors. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freemann and Co., 1979.

[GN96] Dirk Grunwald and Richard Neves. Whole-Program Optimization for Time and Space Efficient Threads. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 50–59, October 1996.

[Gru91] Dirk Grunwald. A Users Guide to AWESIME: An Object Oriented Parallel Programming and Simulation System. Technical Report CU-CS-552-91, University of Colorado at Boulder, November 1991.

[GSSD00] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren. Architecture and Design of AlphaServer GS320. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages –, November 2000.

[GU96] Winfried Grünewald and Theo Ungerer. Towards Extremely Fast Context Switching in a Block-Multithreaded Processor. In *Proceedings of the 22nd Euromicro Conference*, pages 592–599, September 1996.

[GU97] Winfried Grünewald and Theo Ungerer. A Multithreaded Processor Designed for distributed Shared Memory Systems. In *Proceedings of the International Conference on Advances in Parallel and Distributed Computing*, pages 209–213, March 1997.

[Gwe96] Linley Gwennap. Digital 21264 Sets New Standard. *Microprocessor Report*, 10(14):11–16, October 1996.

[Gwe98] Linley Gwennap. Alpha 21364 to Ease Memory Bottleneck. *Microprocessor Report*, 12(14):12–15, October 1998.

[Han93] Jim Handy. *The Cache Memory Handbook*. Academic Press, 1993.

[Han96] Craig Hansen. MicroUnity's MediaProcessor Architecture. *IEEE Micro*, 16(4):34–41, August 1996.

[HB93] Matthew Haines and Wim Böhm. An Evaluation of Software Mulitthreading in a Conventional Distributed Memory Multiprocessor. In *Proceedings of the Symposium on Parallel and Distributed Systems*, pages 106–113, December 1993.

[HB94] Roger Hockney and Michael Berry. Public International Benchmarks for Parallel Computers. *Scientific Programming*, 3(2):101–146, Summer 1994.

[HCC89] Wen-mei W. Hwu, Thomas M. Conte, and Pohua P. Chang. Comparing Software and Hardware Schemes for Reducing the Cost of Branches. In *Proceedings of the 16th International Symposium on Computer Architecture (ISCA)*, pages 224–233, May 1989.

[HCM94] Matthew Haines, David Cronk, and Piyush Mehrotra. On the Design of Chant: A Talking Threads Package. In *Proceedings Supercomputing*, pages 350–359, November 1994.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Publishers B.V. (North-Holland), 1977.

[Hey91] A. J. G. Hey. The Genesis Distributed-Memory Benchmarks. *Parallel Computing*, 17(10–11):1275–1283, December 1991.

[HF88] Robert H. Halstead and Tetsuya Fujita. MASA: A Multithreaded Processor Architecture for Parallel Symbolic computing. In *Proceedings of the 15th International Conference on Computer Architecture (ISCA)*, pages 443–451, May 1988.

[HJ91] John L. Hennessy and Norman P. Jouppi. Computer Technology and Architecture: An Evolving Interaction. *IEEE Computer*, 24(9):18–29, September 1991.

[HKN+92] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, and Teiji Nishizawa. An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads. In *Proceedings of the 19th International symposium on Computer Architecture (ISCA)*, pages 136–145, May 1992.

[HMB+93] Richard E. Hank, Scott A. Mahlke, Roger A. Bringmann, John C. Gyllenhaal, and Wen-mei W. Hwu. Superblock Formation Using Static Program Analysis. In *Proceedings of the 26th International Symposium on Microarchitecture*, pages 247–255, December 1993.

[HMC+93] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Oulette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *Journal of Supercomputing*, 7(1–2):229–248, May 1993.

[HmMS98] Mark Horowitz, Margaret martonosi, Todd C. Mowry, and Michael D. Smith. Informing Memory Operations: Memory Performance Feedback Mechanisms and Their Applications. *ACM Transactions on Computer Systems*, 16(2):170–205, May 1998.

[Hoc93] Roger W. Hockney. Performance parameters and benchmarking of supercomputers. In *Computer Benchmarks*, Advances in Parallel Computing, pages 41–64. Elsevier Science Publishers, 1993.

[Hol78] W. R. Holland. The role of mesoscale eddies in the general circulation of the ocean. *Journal of Physical Oceanography*, 8(3):363–392, May 1978.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2nd. edition, 1996.

[HU75] Matthew S. Hecht and Jeffrey D. Ullman. A Simple Algorithm for Global Data Flow Problems. *SIAM Journal of Computing*, 4(4):519–532, December 1975.

[IEE85] IEEE Standard for Binary Floating-Point Arithmetic, 1985.

[Ita01] Intel Corp. *Intel Itanium Architecture Software Developers Manual*, 2001.

[Jou90] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*, pages 364–373, May 1990.

[Kae00] Daniel Kaestner. PROPAN: A Retargetable System for Postpass Optimisations and Analyses. In *Proceedings of the ACM Workshop on Languages, Compilers and Tools for Embedded Systems*, pages –, June 2000.

[Kan95] Gerry Kane. *PA-RISC 2.0 Architecture*. Prentice-Hall, 1995.

[KCA91] Kiyoshi Kurihara, David Chaiken, and Anant Agarwal. Latency Tolerance through Multithreading in Large-Scale Multiprocessors. In *Proceedings of International Symposium on Shared Memory Multiprocessing*, pages 341–361, April 1991.

[KCO+94] Ravi Konuru, Jeremy Casas, Steve Otto, Robert Prouty, and Jonathan Walpole. A User-Level Process Package for PVM. In *Proceedings of the Scalable High Performance Computer Conference*, pages 48–55, May 1994.

[KD92] Stephen W. Keckler and William J. Dally. Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism. In *Proceedings of the 19th International Conference on Computer Architecture (ISCA)*, pages 202–213, May 1992.

[KDM+98] Stephen W. Keckler, William J. Dally, Daniel Maskit, Nicholas P. Carter, Andrew Chang, and Whay S. Lee. Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor. In *Proceedings of the 25th International Conference on Computer Architecture (ISCA)*, pages 306–317, June 1998.

[KE91] D. R. Kaeli and P. G. Emma. Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, pages 34–42, May 1991.

[Kes99] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March 1999.

[KFW94] R. Kent Koeninger, Mark Furtney, and Martin Walker. A Shared Memory MPP from Cray Research. *Digital Technical Journal*, 6(2):8–21, Spring 1994.

[KH92] Gerry Kane and Joe Heinrich. *MIPS RISC architecture*. Prentice-Hall, 1992.

[Kil73] Gary A. Kildall. A Unified Approach to Global Program Optmization. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 194–206, October 1973.

[KMW98] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 Microprocessor Architecture. In *Proceedings International Conference on Computer Design (ICCD)*, pages 90–95, October 1998.

[KN95] Michael Kantrowitz and Lisa M. Noack. Functional Verification of a Multiple-issue, Pipelined, Superscalar Alpha Processor - the Alpha 21164 CPU chip. *Digital Technical Journal*, 7(1):136–144, Winter 1995.

[KS88] James T. Kuehn and Burton J. Smith. The Horizon Supercomputing System: Architecture and Software. In *Proceedings Supercomputing*, pages 28–34, November 1988.

[KU75] J. B. Kam and Jeffrey D. Ullmann. Monotone Data Flow Analysis Frameworks. Technical Report No. 169, EE Department, Princeton University, 1975.

[Lam99] Monica S. Lam. An Overview of the SUIF2 System. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages –, May 1999.

[LB96] Mat Loikkanen and Nader Bagherzadeh. A Fine-Grain Multithreading Superscalar Architecture. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 163–168, October 1996.

[LEL+99] Jack L. Lo, Susan J. Eggers, Henry M. Levy, Sujay S. Parekh, and Dean M. Tullsen. Tuning Compiler Optimizations for Simultaneous Multithreading. *International Journal of Parallel Programming*, 27(6):477–503, November 1999.

[Lev83] Bruce W. Leverett. *Register Allocation in Optimizing Compilers*. UMI Research Press, 1983.

[LFA96] David K. Lowenthal, Vincent W. Freeh, and Gregory R. Andrews. Using Fine-Grain Threads and Run-Time Decision Making in Parallel Computing. *Journal of Parallel and Distributed Computing*, 37(1):41–54, August 1996.

[LGH94] James Laudon, Anoop Gupta, and Mark Horowitz. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 308–318, October 1994.

[LH86] Tomas Lang and Miquel Huguet. Reduced Register Saving/Restoring in Single-Window Register Files. *Computer Architecture News*, 14(3):17–26, June 1986.

[LM96] Chi-Keung Luk and Todd C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the 7th International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 222–233, October 1996.

[LMB92] Hohn R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly Associates, Inc., 2nd. edition, 1992.

[LO96] Mike Loukides and Andy Oram. *Programming with GNU Software*. O'Reilly Associates, Inc., 1996.

[LR97] Daniel Leibholz and Rahul Razdan. The Alpha 21264: A 500 MHz Out-of-Order Execution Microprocessor. In *Proceedings Computer Conference (COMPCON)*, pages 28–36, February 1997.

[LS84] J. K. L. Lee and A. J. Smith. Branch Prediction Strategies and Branch Target Buffer Design. *IEEE Computer*, 17(1):6–22, January 1984.

[LS95] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[LW00] K. S. Loh and W. F. Wong. Multiple context multithreaded superscalar processor architecture. *Journal of Systems Architecture*, 46(3):243–258, January 2000.

[MBB+98] M. Matson, D. Bailey, S. Bell, L. Biro, S. Butler, J. Clouser, J. Farrell, M. Gowan, D. Priore, and K. Wilcox. Circuit Implementation of a 600 MHz Superscalar RISC Microprocessor. In *Proceedings International Conference on Computer Design (ICCD)*, pages 90–95, October 1998.

[MBC+94] Dina L. McKinney, Masooma Bhaiwala, Kwong-Tak A. Chui, Christopher L. Houghton, James R. Mullens, Daniel L. Leibholz, Sanjay J. Patel, Delvan A. Ramey, and Rosenbluth Mark B. Digital's DECchip 21066: The First Cost-focused Alpha AXP Chip. *Digital Technical Journal*, 6(1):66–77, Winter 1994.

[McC95] John D. McCalpin. Sustainable Memory Bandwidth in Current High Performance Computers. http://home.austin.rr.com/mccalpin/papers/bandwidth/bandwidth.html, 1995.

[McF93] Scott McFarling. Combining Branch Predictors. Technical Report WRL TN-36, Digital Western Research Laboratory (WRL), June 1993.

[MCL98] Todd C. Mowry, Charles Q. C. Chan, and Adley K. W. Lo. Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 300–311, January 1998.

[McM88] Frank H. McMahon. The Livermore Fortran Kernels Test of the Numerical Performance Range. In *Performance Evaluation of Supercomputers*, pages 143–186. Elsevier Science Publishers B.V. (North-Holland), 1988.

[MD96] A. Mikschl and W. Damm. MSparc: A Multithreaded Sparc. In *Proceedings of the 2nd International Euro-Par Conference*, pages 461–469, August 1996.

[ME97] Soo-Mook Moon and Kemal Ebcioğlu. Parallelizing Nonnumerical Code with Selective Scheduling and Software Pipelining. *ACM Transactions on Programming Languages and Systems*, 19(6):853–898, November 1997.

[MH86] Scott McFarling and John Hennessy. Reducing the Cost of Branches. In *Proceedings of the 13th International Symposium on Computer Architecture (ISCA)*, pages 396–403, May 1986.

[MHL91] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. An Efficient Method for Exact Dependence Analysis. In *Proceedings of the 1991 Symposium on Programming Languages Design and Implementation*, pages 1–14, July 1991.

[MJA+01] Robert O. Mueller, A. Jain, W. Anderson, T. Benninghoff, D. Bertucci, J. Burnette, T. Chang, J. Eble, R. Faber, D. Gowda, J. Grodstein, G. Hess, J. Kowaleski, A. Kumar, B. Miller, P. Paul, J. Pickholtz, S. Russell, M. Shen, T. Truex, A. Varadharajan, D. Xanthopoulos, and T. Zou. A 1.2 GHz Alpha Microprocessor. In *Proceedings International Solid State Circuits Conference (ISSCC)*, pages –, February 2001.

[MLC+92] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, December 1992.

[MNL96] Gail C. Murphy, David Notkin, and Erica S.-C. Lan. An Empirical Study of Static Call Graph Extractors. In *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, pages 90–99, March 1996.

[MR96] Michael Metcalf and John Reid. *Fortran 90/95 explained*. Oxford University Press, 3rd. edition, 1996.

[MR99] Todd C. Mowry and Sherwyn R. Ramkissoon. Software-Controlled Multithreading Using Informing Memory Operations. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 121–132, January 1999.

[MSLM91] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-Class User-Level Threads. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 110–121, October 1991.

[MSSW94] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC architecture: A Specification for a new Family of RISC Processors*. Morgan Kaufmann Publishers, 2nd. edition, 1994.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design & Integration*. Morgan Kaufmann Publishers, 1997.

[Mue93] Frank Mueller. A Library Implementation of POSIX Threads under UNIX. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 29–41, January 1993.

[NL92] Jason Nieh and Marc Levoy. Volume Rendering on Scalable Shared Memory MIMD Architectures. In *Proceedings of the Boston Workshop on Volume Visualization*, pages 17–24, October 1992.

[NS97] Richard Neves and Robert B. Schnabel. Threaded Runtime Support for Execution of Fine Grain Parallel Code on Coarse Grain Multiprocessors. *Journal of Parallel and Distributed Computing*, 42(2):128–142, May 1997.

[Oed96] Wilfried Oed. *Cray Research Massiv-paralleles Prozessorsystem Cray T3E*. Cray Research GmbH, November 1996.

[PD80] David A. Patterson and David R. Ditzel. The Case for the Reduced Instruction Set Computer. *Computer Architecture News*, 9(6):25–38, September 1980.

[PE96] James Philbin and Jan Edler. Very Lightweight Threads. In *Proceedings of the 1st International Workshop on High-Level Programming Models and Supportive Enviroments*, pages 95–104, April 1996.

[PEA+96] James Philbin, Jan Edler, Otto J. Anshus, Doug C. Douglas, and Kai Li. Thread Scheduling for Cache Locality. In *Proceedings of the 7th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 60–71, October 1996.

[PG99] Joan-Manuel Parcerisa and Antonio Gonzalez. The Synergy of Multithreading and Access/Execute Decoupling. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 59–63, January 1999.

[PH98] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, 1998.

[PSR92] Shien-Tai Pan, Kimming So, and Joseph T. Rahmeh. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 76–84, October 1992.

[PW91] R. Guru Prasadh and Chuan-lin Wu. A Benchmark Evaluation of a Multi-Threaded RISC Processor Architecture. In *Proceedings of the International Conference on Parallel Processing*, pages 84–91, August 1991.

[RG94] Edward Rothberg and Anoop Gupta. An Efficient Block-Oriented Approach to Parallel Sparse Cholesky Factorization. *SIAM Journal of Scientific Computing*, 15(6):1413–1439, November 1994.

[RGSL96] John Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler. In *Proceedings of the 1996 Symposium on Programming Languages Design and Implementation*, pages 1–11, July 1996.

[Roc00] Daniel N. Rockmore. The FFT: An Algorithm The Whole Family Can Use. *IEEE Computing in Science & Engineering*, 2(1):60–64, January 2000.

[SAB+98] S. Storino, A. Aippersbach, J. Borkenhagen, R. Eickemeyer, S. Kunkel, S. Levenstein, and G. Uhlmann. A Commercial Multi-threaded RISC Processor. In *Proceedings of the International Sold-State Circuits Conference (ISSCC)*, pages 234–235, February 1998.

[SBCvE90] Rafael H. Saavedra-Barrera, David E. Culler, and Thorsten von Eicken. Analysis of Multithreaded Architectures for Parallel Computing. In *Proceedings 2nd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 169–178, July 1990.

[Sco96] Steven L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 26–36, October 1996.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 196–205, June 1994.

[Sei00] Rene Seindal. GNU m4 - A Powerful Macro Processor . http://www.seindal.dk/rene/gnu/man/m4_toc.html, 2000.

[SG98] Avi Silberschatz and Peter Galvin. *Operating System Concepts*. Addison-Wesley Longman, Inc., 5th edition, 1998.

[SGL92] Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[SH92] Jaswinder Pal Singh and John L. Hennessy. Finding and Exploiting Parallelism in an Ocean Simulation Program: Experience, Results, and Implications. *Journal of Parallel and Distributed Computing*, 15(1):27–48, May 1992.

[SHHG93] Jaswinder Pal Singh, Chris Holt, John L. Hennessy, and Anoop Gupta. A Parallel Adaptive Fast Multipole Method. In *Proceedings Supercomputing 93*, pages 54–65, November 1993.

[SHT+95] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Annop Gupta, and John L. Hennessy. Load Balancing and Data Locality in Adaptive Hierarchical n-Body Methods: Barnes-Hut, Fast Multipole, and Radiosity. *Journal of Parallel and Distributed Computing*, 27(2):118–141, June 1995.

[Sim00] Dezsö Sima. The Design Space of Register Renaming Techniques. *IEEE Micro*, 20(5):70–83, September 2000.

[Sin93] Jaswinder Pal Singh. *Parallel Hierarchical N-Body Methods and their implications for multiprocessors*. PhD thesis, Stanford University, February 1993.

[Sit92] Richard L. Sites. Alpha AXP Architecture. *Digital Technical Journal*, 4(4):19–34, Fall 1992.

[SJH89] Michael D. Smith, Mike Johnson, and Mark A. Horowitz. Limits on Multiple Instruction Issue. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 290–302, April 1989.

[SKMR92] Anton Sites, Richard L.and Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Digital Technical Journal*, 4(4):137–152, Fall 1992.

[Smi78] Burton J. Smith. A Pipelined, Shared Resource MIMD Computer. In *Proceedings of the Conference on Parallel Processing*, pages 6–8, August 1978.

[Smi81a] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Proceedings SPIE Real Time Signal Processing IV*, pages 241–247, August 1981.

[Smi81b] J. E. Smith. A Study of Branch Prediction Strategies. In *Proceedings of the 8th International Symposium on Computer Architecture (ISCA)*, pages 135–148, May 1981.

[Smi82] James E. Smith. Decoupled Access/Execute Computer Architectures. In *Proceedings of 9th International Symposium on Computer Architecture (ISCA)*, pages 112–119, April 1982.

[SOHL+98] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI - The Complete Reference*, volume 1. MIT Press, 1998.

[ST96] Steven L. Scott and Gregory M. Thorson. The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus. In *Proceedings HOT Interconnects IV*, pages –, August 1996.

[Sto93] Harold S. Stone. *High-performance computer architecture*. Addison-Wesley, 1993.

[Str78] W. D. Strecker. VAX-11/780 - A virtual address extension to the DEC PDP-11 family. In *Proceedings AFIPS National Computer Conference (NCC)*, pages 967–980, July 1978.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd. edition, 1997.

[SW93] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, March 1993.

[TC88] Robert H. Thomas and Will Crowther. The Uniform System: An approach to runtime support for large scale shared memory parallel processors. In *Proceedings of the International Conference on Parallel Processing*, pages 245–254, August 1988.

[TEE+95] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of*

the 22nd International symposium on Computer Architecture (ISCA), pages 191–202, June 1995.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd International symposium on Computer Architecture (ISCA)*, pages 392–403, June 1995.

[Tho70] J. E. Thornton. *Design of a Computer: The Control Data 6600*. Scott, Foresman and Co., 1970.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[Tru96] Digital Equipment Corporation. *Digital Unix Calling Standard for Alpha Systems*, 1996.

[Tru01] Compaq Computer Corporation. *Tru64 UNIX Operating System Version 5.1A QuickSpecs*, 2001.

[TS88] Mark R. Thistle and Burton J. Smith. A Processor Architecture for Horizon. In *Proceedings Supercomputing*, pages 35–41, November 1988.

[VA00] Vladimir Vlassov and Rassul Ayani. Analytical modeling of multithreaded architectures. *Journal of Systems Architecture*, 46(13):1205–1230, November 2000.

[vdSdR93] A. J. van der Steen and P. P. M. de Rijk. Guidelines for the use of the EuroBen Benchmark. Technical Report TR3, The EuroBen Group, February 1993.

[vRU99] Jurij Šilc, Borut Robič, and Theo Ungerer. *Processor Architecture*. Springer Verlag, 1999.

[Wal86] David W. Wall. Global Register Allocation at Link Time. In *Proceedings of the ACM Symposium on Compiler Construction*, pages 264–275, June 1986.

[Wal92] David W. Wall. Limits of Instruction-Level Parallelism. In *Proceedings 4th International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 176–188, April 1992.

[Wal94] David W. Walker. The design of a standard message passing interface for distributive memory concurrent computers. *Parallel Computing*, 20(4):657–673, March 1994.

[WDH89] Mark Weiser, Alan Demers, and Carl Hauser. The Portable Common Runtime Approach to Interoperability. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 114–122, December 1989.

[Wei80] William E. Weihl. Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables and Label Variables. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages (POPL)*, pages 83–94, January 1980.

[WFW+94] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[WG93] David L. Weaver and Tom Germond. *The Sparc Architecture Manual Version 9*. Prentice-Hall, 1993.

[Wie82] Cheryl A. Wiecek. A Case Study of VAX-11 Instruction Set Usage For Compiler Execution. In *Proceedings Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 177–184, March 1982.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and

Methodological Consideration. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, May 1995.

[WSH94] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The Performance Advantage of Integrated Block Data Transfer in Cache-Coherent Multiprocessors. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 219–229, October 1994.

[WW93] Carl A. Waldspurger and William E. Weihl. Register Relocation: Flexible Contexts for Multithreading. In *Proceedings of the 20th International symposium on Computer Architecture (ISCA)*, pages 120–130, June 1993.

[YP92] Tsu-Yu Yeh and Yale N. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)*, pages 124–134, May 1992.

[YP93] Tsu-Yu Yeh and Yale N. Patt. A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. In *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*, pages 257–266, May 1993.