Master’s Thesis in Electrical Engineering

Luleå University of Technology

A Platform for Evaluation of Multiprocessors in Throughput-Oriented Systems

Fredrik Warg

Abstract

Multiprocessors are widely used today as a way to achieve high performance using commodity microprocessors. Since different applications place different demands on the system, performance evaluation in the design phase of such systems is important. The most common method is the use of simulators.

This thesis presents a simulator of a multiprocessor for throughput-oriented applications, using a distribution-driven simulation method. With this method hardware, and to some extent software, parameters can be studied. A case study is performed, focusing on performance issues for various characteristics of a multiprocessor implementation of a telecom server that today is a uniprocessor.

The results show that the key bottlenecks for this particular system lies in parallelisation of the software, which is written with a uniprocessor environment in mind. Also, providing sufficient bus bandwidth is an important factor. Other issues investigated, such as scheduling techniques, three different architectural choices, and degree of memory interleaving, is found to have a smaller impact on performance.

Preface

This Master's Thesis is the final part required for my Master of Science degree in Electrical Engineering / Computer Engineering at Luleå University of Technology (LTU). The work has been carried out during the spring of 1999, at Chalmers University of Technology (CTH), and is a part of a NUTEK sponsored project involving the High Performance Computer Architecture group at CTH, and Ericsson Utveckling AB (UAB). The aim of the project is to investigate, and find ways to improve, performance of multiprocessors used in transaction-oriented systems.

Acknowledgements

I would like to thank my supervisor at CTH, Fredrik Dahlgren, for advice and encouragement; Halim Smai and Lars-Åke Johansson at UAB for providing data for the case study, ideas, and feedback on the report; and my examiner at LTU, Dennis M. Akos.

Fredrik Warg

Göteborg, May 1999

Table of Contents

1 Introduction

The use of multiprocessors as an approach for boosting performance has become increasingly popular. Today, shared memory multiprocessors using general-purpose microprocessors are available from all of the major computer vendors. The performance of such a system is, to a significant degree, dependent on the design of the memory system. It is important to realize, though, that the application plays an important role; the resource usage varies between different classes of applications. This fact should be taken into account when designing the system. There is no "one true way" that always results in optimal performance. Therefore, it is important to make a thorough evaluation of the memory system requirements before building a multiprocessor. The preferred method for performance evaluation today is the use of simulators, since they are more flexible and cost-efficient than building prototypes, and more accurate than other known methods.

The focus of this work is performance evaluation of throughput-oriented systems. Here, a throughput-oriented system is defined as a system where each program is executed on its own processor in a multiprocessor system; the throughput of the system as a whole is measured as the sum of the throughput of each of the processors. The workload of interest is transaction processing, which is used in an important group of applications, such as database management systems and telecom servers.

In addition to the impact of hardware characteristics, performance is affected by the parallelism one can extract from existing software. Most software today is not written explicitly to run on a multiprocessor. Therefore, multiprocessors are typically built so that the parallel processing is completely transparent to the application programs. With this in mind, both scheduling of transactions onto the processors, and data dependencies in the software that impose restrictions on the execution should be taken into account in order to get useful results from simulations.

Multiprocessor design for throughput-oriented systems has previously been investigated by Chiang and Sohi [4] using an analytical method based on mean value analysis. However, their method is not useful if software related issues, such as scheduling, are to be included in the study. In addition, the method is request-oriented and therefore not suitable for bottleneck analysis; it is not obvious from the results where the delays are incurred. The method presented here address these issues.

The most flexible and accurate simulation method, and one that is becoming increasingly popular, is program-driven simulation. However, for some applications, such as telecom applications, the complexity of the system, as well as the difficulty involved in determining typical application scenarios, makes it difficult to develop program-driven simulators. In order to gain an understanding of the factors involved in building a multiprocessor system, it can be useful to build a less complex simulator that does not require complete instruction-level simulation and a real workload, but is still sufficiently detailed to include important hardware and software related issues.

This thesis presents a simulator using distribution-driven workload based on information on a program function level, including which functions are executed and their execution times, for a number of transactions. With this data, it is possible to obtain a measure on throughput for the system. The simulator also facilitates investigation of some software scheduling and execution schemes. Cache misses, coherency traffic, write back traffic, and cache affinity effects, as well as bus contention, queuing delays and memory latencies are taken into account by the simulator. For this, additional data on footprint and read/write frequencies for the software is needed. A case study evaluating performance issues when converting a telecom system to a multiprocessor is performed. For the case study, the function level data has been captured from a real telecom server.

The results show that the parallelisation of the software is important for the performance when increasing the number of processors. The difference between the worst of the two schemes examined in this report and a "best case" is 24% for an 8 processor system. Furthermore, it is important to make sure bus bandwidth is sufficient, or else performance drops of quickly.

The rest of this report is organized as follows: Section 2 provides background information on methodology as well as both the system used in the case study and the target system. Next, Section 3 describes the design of the simulator, including its input and output, and Section 4 shows simulation results from the case study. Finally, conclusions are drawn in Section 5.

2 Background

This section will give some background that explains design decisions, and introduce some concepts necessary to understand the rest of the report. Section 2.1 describes different methodologies used to evaluate the performance of architectural decisions, 2.2 describes the program system under examination, and 2.3 the target architecture that will be the focus of the simulations. Finally, Section 2.4 discusses how to measure performance.

2.1 Overview of Performance Evaluation Methodologies

There are three possible approaches for performance evaluation of computer architectures: simulation, building analytical models, or prototype construction. Prototype construction is time-consuming and expensive, and therefore something that is generally avoided until a late stage of product development. For broader architectural studies, other methods are preferred. This section discusses the merits and problems with different simulation techniques and analytical models.

Analytical methods have been explored primarily because of their speed. For example, several methods based on mean value analysis have been developed [4] [9] [14]. Analytical methods are indeed very fast, but usually less accurate than simulation, and does not allow for the possibility of trying different software scheduling and execution schemes.

The simulation methodologies can be classified into three groups according to how the workload is simulated. Program-driven simulators use real programs as their workload, either as unmodified binaries or in a format specific to the simulator. Trace-driven simulators use traces of input events, such as memory references, captured on an existing system, but do not execute any code by itself. Distribution-driven workloads make use of random variables of, for instance, memory references, locality, and program behavior. Typically, the stochastic parameters are based on statistics from a reference system.

A simulator with program-driven load mimics the target system closely. Real application programs are executed on top of the simulator. The advantage is flexibility in target system models and accurate results for the applications used in the simulations.

A common type of program-driven simulator is the instruction-set simulator, which executes every instruction in the workload exactly as the target machine would. This approach can be extended to cover the entire execution environment, which includes all devices an application expects to find, such as CPU, MMU, and SCSI devices. This makes it possible to run an entire operating system on top of the simulator, and include OS activity in the evaluations, as well as run unmodified programs. The inclusion of OS or system software activity is important, as it can have a significant impact [3]. This approach is used in [2] [10] [11]. A weakness is that such a simulator tend to be computationally expensive, and even if efficient instruction-set simulators exist today, it is not always practical to construct one. The construction of such a simulator is often extremely complicated, due to the need to recreate the entire execution environment. This is especially true for complex program systems including system software.

A particular type of simulator using program-driven workload is an execution-driven simulator. Instead of emulating the instruction set as the instruction-set simulation approach does, the code is executed on a real processor. The execution is interrupted at predefined interaction points (such as access to shared data or message passing, this depends on the target architecture), where the simulator takes over. Thus, the computations are executed on a real processor, while the memory system is simulated. The advantage of this approach is speed, but it is considerably less flexible than instruction-level simulators. For instance, the simulator can only be used on a system with the same instruction set as the target system, which can be problematic, especially when evaluating a future system where the target instruction set has not been implemented in hardware yet. The execution-driven approach has been used in [5].

A simulator with trace-driven workload use memory reference traces generated by programs running on an existing system as input to the simulator. This works well as long as the target system is fairly similar to the system the traces were captured from. Using traces from a uniprocessor on a multiprocessor system simulation is more difficult as, for example, execution order will be different on the target system. Common methods for trace-driven simulation on uniprocessors are summarized in a survey by Uhlig and

Mudge [12]. Different methods for multiprocessor trace generation have been proposed [7] [1], but are generally considered less accurate than program-driven methods.

A distribution-driven workload is based on statistics on system behavior. Simulation is fast, and it is easy to tweak system parameters in order to see what would happen in various application scenarios. The difficulty lies in obtaining accurate parameters, and include in the model how these parameters interact.

The primary advantage of distribution-driven and analytical models is that no code execution is performed, and thus the complexity of correctly modeling an instruction-set, and all devices needed to create the execution environment necessary to run real programs, is avoided. The latter is a very demanding and time-consuming task, and the more hardware-devices and special-purpose chips there are, the more complex the simulator gets. A simulator using the simpler techniques can be built with less effort for use in a first round of architectural evaluations.

In this work, a distribution-driven method will be presented.

2.2 Description of the Modeled System

The system under examination in the case study is the processor server of a telecom system. The throughput in this system is measured in transactions, or requests for service, generated by an external source. The requests vary in number and type depending on the users (i.e. all the users of the public telephone system). Therefore, the load of the system is non-deterministic and can vary over time within wide bounds. A transaction is triggered by a message from the external source and is considered to be completed when all execution spawned by the request is completed.

The messages originating from outside the system are queued by the scheduling processor (SP) until they can be dispatched to the execution processor (EP). A transaction typically requires a number of program functions to be executed. Some of the functions will generate message transfers within the system, and these messages are also handled by the SP (see Figure 1).

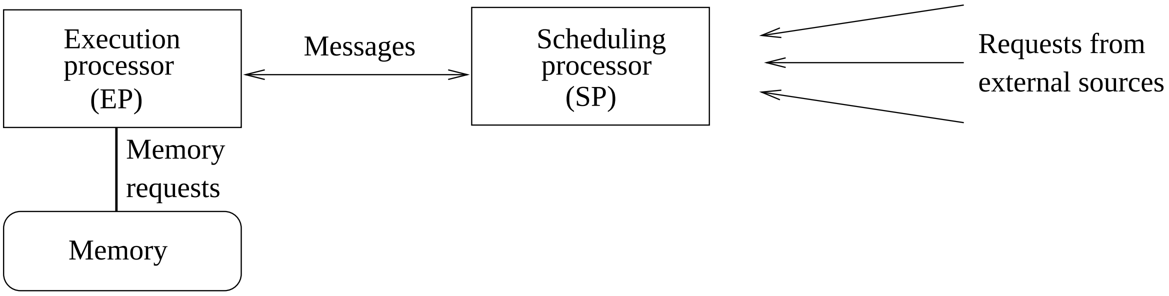

Figure 1. The target system receives messages through the scheduling processor. The programs are executed in the execution processor.

The transactions can be divided into one or more uninterrupted sequences of program functions. Such a sequence will be called transaction subtask. A subtask is initiated by a message from the SP to the EP. The messages generated by functions within a subtask might give rise to new subtasks (see Figure 2).

This system currently exists as a uniprocessor[1]. The objective of this study is to find performance bottlenecks of the system in a multiprocessor configuration. To that end, different hardware architectures as well as different software execution schemes will be evaluated using a custom-built simulator.

2.2.1 Parallelism

The software that runs on this system is designed for use on a uniprocessor. If unmodified code from the uniprocessor is to be used on a new multiprocessor, it is important to preserve the original programming model, so that old code, even if it is badly written, under no circumstances can produce different results on the new system when compared to the old. Three different execution schemes will be included in

[1] At least it is a uniprocessor with respect to code execution, which only takes place in the EP. The SP acts as a coprocessor, serving the EP with new work.

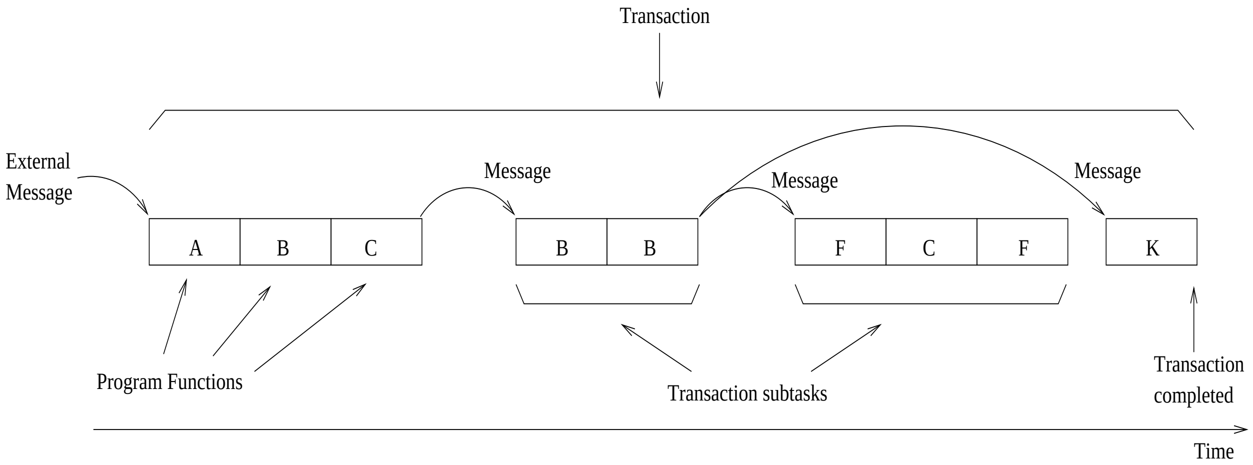

Figure 2. The relationship between transactions, transaction subtasks, functions, and messages. Note that the messages are passed from the EP to the SP, and the next subtask is not run until it is scheduled by the SP. The functions can be used repeatedly within the same subtask; they are also not specific to a subtask. The same function can be used in different subtasks and transactions.

the study, each based on different assumption of the programming model. The schemes that use mutual exclusion mechanisms (A and B) both assume that function allocation is done in run-time, as every new function is encountered. The purpose of this evaluation is to see how much parallelism is lost for scenarios that deviate from the ideal model. The schemes are:

- *Ideal execution scheme*: Any function can be executed at any time on any EP. There are no obstacles limiting parallelism.
- *Execution scheme A*: The simultaneous execution of the same function on two processors is disallowed; this situation can occur since some functions are used frequently, and in many types of transactions and subtasks. This scheme is necessary if functions use variables of which there exists only one instance (i.e. there is not a separate copy of the variable for every invocation of the function). This could lead to data inconsistency problems since the two processors might overwrite each others data at any time. A solution to the problem is to treat every function as a critical region, and enforce mutual exclusion.
- *Execution scheme B*: In addition to the mutual exclusion described in *A*, we add the assumption that functions rely on exclusive access to its variables during an entire transaction subtask, since a function may be visited several times during the subtask. Therefore, once a function is locked (made exclusive for an EP), it is not released until the end of the current subtask. However, this scheme is deadlock-prone, which is illustrated in Figure 3.

Since deadlocks can occur in Execution scheme B, a mechanism to deal with deadlocks is necessary. Deadlocks can occur in a system where four criteria are fulfilled: resources can not be shared (mutual exclusion), a resource can not be revoked after is has been locked (no resource preemption), it is possible to hold a resource while waiting for another, and finally that it is possible for a cyclic request pattern to emerge. There are three possible strategies to deal with the deadlock problem. Which one to choose is a matter of performance and implementation feasibility. The first approach is deadlock prevention, or to make sure deadlocks can not occur by voiding one of the criteria stated above. The second is avoidance, or some way to dynamically make sure that the system will not enter an unsafe state (a state in which deadlock can occur) before granting lock on a resource. The third method is deadlock detection, which relies on software to detect a deadlock after it happened followed by some way to resolve the deadlock [13].

The scheme investigated in this study will be based on deadlock detection. If there are only two processors, as in Figure 3, deadlock detection is fairly simple, we only need to detect if the two processors are

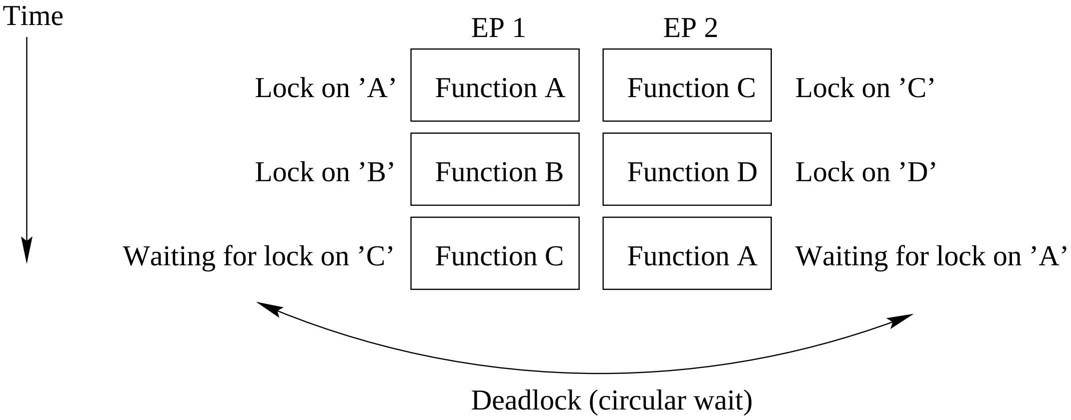

Figure 3. EP1 and EP2 executes different subtasks, possibly belonging to different transactions, but using the same functions. In this case, EP1 have locks on functions A and B, while EP2 have locks on C and D. Neither EP will release its locks until the subtask is completed. However, EP1 is waiting to get a lock on function C, held by EP2, and EP2 is waiting on function A, locked by EP1. Unless this circular wait is detected and resolved, both EPs will be stalled forever.

waiting for each other. As illustrated in Figure 4, however, the situation is more difficult with additional processors. In Figure 4D, a loop is formed, locking processors 1-3 in circular wait. EP 4 is also deadlocked, even if it is not in the loop. The deadlock detection mechanism must be able to find such circular wait loops with an arbitrary number of processors involved.

Deadlocks will be resolved by rollback, i.e. by restarting one of the subtasks causing the deadlock. The victim's resources will be released as a part of the rollback, one of the resources will be grabbed by a waiting processor instead, and the deadlock is resolved. A rollback must also undo all changes that has been made by the partially executed task. The youngest task (having been the shortest time in the system) should be the victim chosen for rollback operations, in order to prevent that one of the tasks suffer from starvation. Starvation could occur if the same task is always restarted, since that might lead to a situation where a task is never allowed to finish. If the youngest task is always chosen, we can guarantee that this is not the case, because a task cannot remain the youngest task forever.

2.2.2 Load Distribution

As mentioned, transactions are queued by the SP and dispatched to the EP when they are first in the queue and the EP is free. In a multiprocessor system, however, scheduling is not as straightforward as in the uniprocessor. To get good performance, the work must be evenly distributed so that all the EPs are utilized as much as possible. Therefore, it becomes necessary to find a good load balancing algorithm that both utilizes the available EPs efficiently, and results in correct execution.

A couple of different scheduling methods will be investigated. The ideal scheduling case is that any subtask can be scheduled to any EP without restrictions, this method will be called "lazy scheduling". However, since subtasks can send new messages to the EP at any time, and assuming that all messages in a transaction must have the same order in a multiprocessor system as in the uniprocessor system (since the messages between subtasks in a transaction can contain data there is again a possible data consistency problem), the implication is that two subtasks that belong to the same transaction must be executed sequentially. That is, a subtask cannot begin to execute before the previous subtask in the same transaction has already finished, otherwise messages could end up in the wrong order (see Figure 5). To solve this problem, one must enforce a scheduling scheme that makes it impossible to execute two subtasks from the same transaction at the same time. However, we still assume that subtasks from different transactions can be executed in any order. Two ways to achieve this will be examined:

- Transaction scheduling method: Scheduling of all subtasks in a transaction onto the same EP, which will automatically result in the correct sequential behavior.

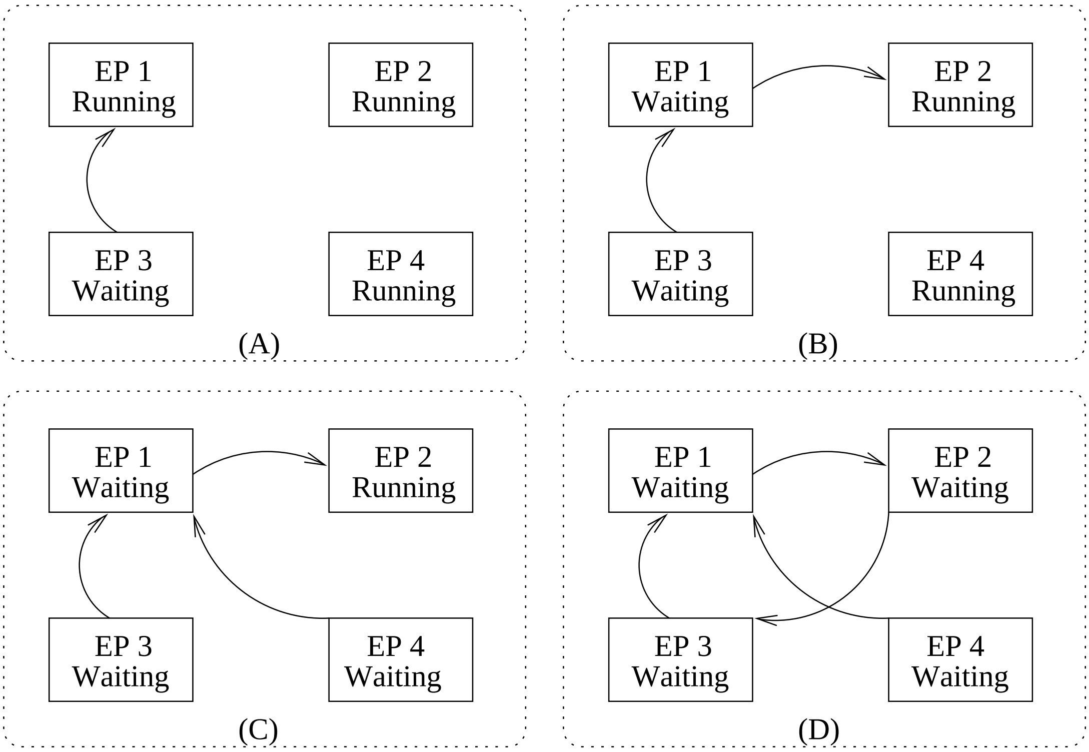

Figure 4. A common wait situation (A) grows (B and C) as more processors need some common resource. Finally, in D, a circular wait loop is formed and the system is deadlocked.

- Central SP method: One central SP handles scheduling for all EPs and detect conflicts. When a conflict is detected, scheduling of the subtask is bypassed in the queue until the conflict is resolved (i.e. the previous subtask in the transaction has finished).

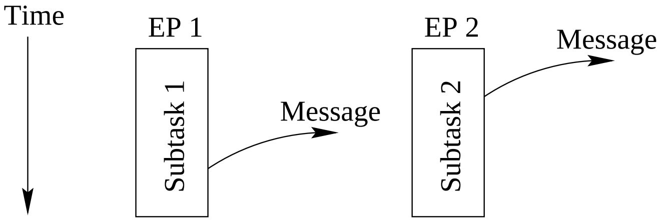

Figure 5. If a message from subtask 2 is sent before the message in subtask 1, there might be a data consistency problem affecting subsequent tasks in the same transaction.

2.3 Target Architecture

The target architecture is a multiprocessor. Each processor is connected to the shared bus via its local two-level cache. Shared memory and I/O units (which in this case are the external sources that initiate transactions) can also be connected to the bus. These types of computers are called shared memory multiprocessors, since all processors can access the same memory. The most common variant of shared-memory multiprocessors use the uniform memory access model, which means that all processors have equal access time to all the memory. Other shared memory architectures are the non-uniform memory model, and the cache only memory model. There are several textbooks describing multiprocessors in more detail, see for instance [6] and [8].

The remainder of this section describes the different units in more detail. Four possible multiprocessor mappings of the modeled system are shown in Figure 6.

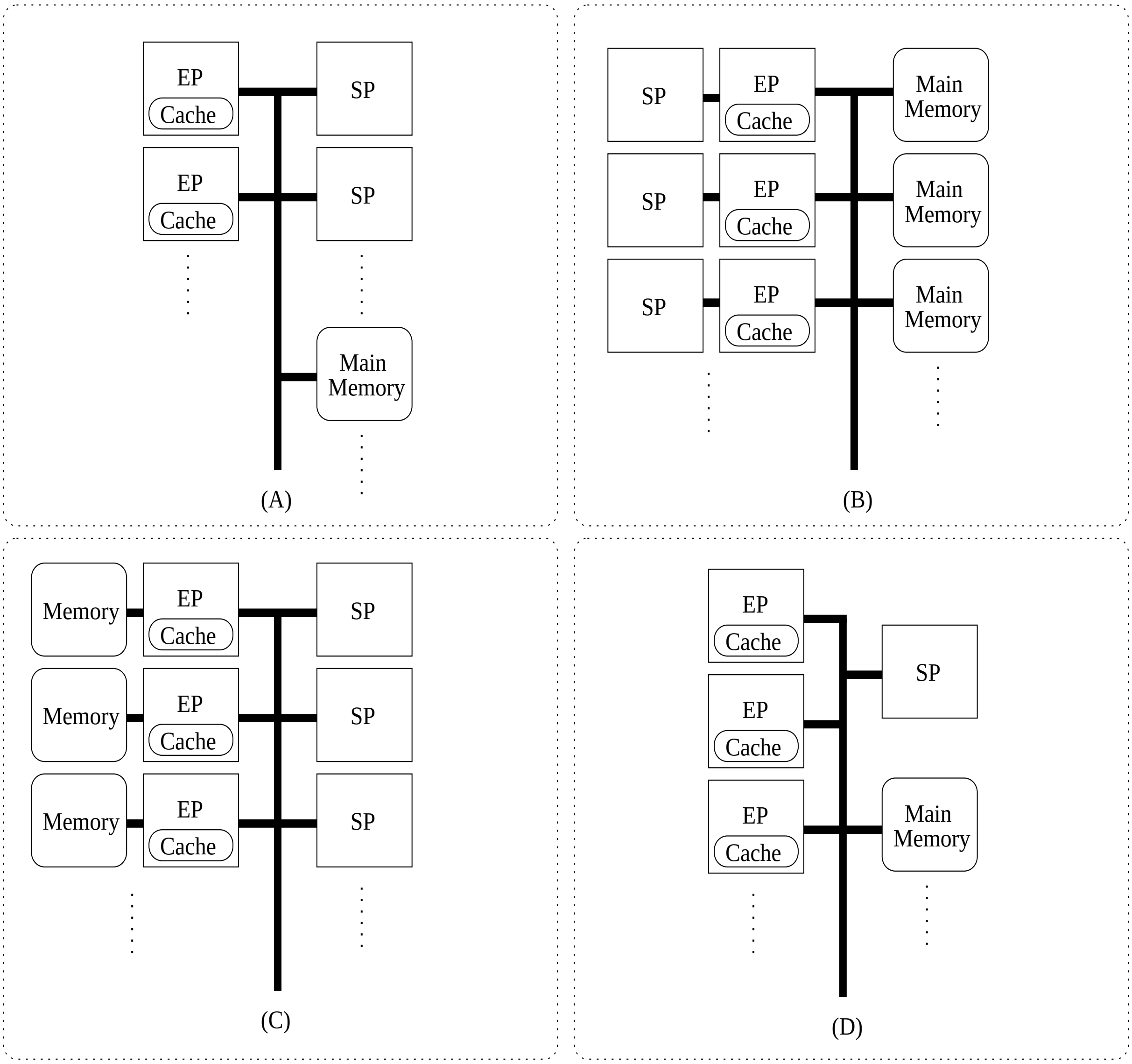

Figure 6. (A) Both EPs and SPs are connected to the shared bus. The main memory is shared, and the number of processors and memory modules is variable. (B) The EPs and SPs are connected directly to one another, like in the uniprocessor system. (C) Each EP has its own local memory. (D) All EPs share a common SP.

2.3.1 Cache Memory

Each processor has a local two-level cache hierarchy (assumed to consist of a level one on-chip cache and a larger level two off-chip cache). The caches are of write-back type. With a write-back cache, modified data is only written back to memory when it does no longer fit in the cache, as opposed to a write-update cache which writes back modified data immediately. The advantage of write-back caches is that bus traffic is usually lower, which is especially important when using a shared bus. Therefore, write-back caches are the most common type in shared bus systems. The disadvantage, compared to the write-update protocol, is that access times to shared data can be longer, because data is only updated when it is accessed, never in advance.

In a multiprocessor, the same data may reside in multiple local caches, as well as in main memory. This gives rise to a data consistency problem. There must be a way to ensure that separate caches cannot contain the same datum but with different values. In order to keep data consistent, a write-invalidate (W-I) snoopy cache coherency protocol will be used. There are several variations of the W-I protocol (for a detailed description of cache coherency protocols, see [6]). The most simple of the W-I protocols, the three-state MSI (Modified-Shared-Invalidated) protocol, will be used in this study. Under the MSI protocol, every cache block can be in one of the following three states:

- *Modified,* means that the local copy of the data has been modified. It also means that none of the other caches has a copy of this data. This data can be modified again without any additional action.
- *Shared,* means that the data is present in at least one other cache. A write to this data must be preceded by an invalidation message, so that the other caches containing the same data can invalidate it. The data can be updated upon confirmation that the other caches have invalidated it (this can be considered to have happened as soon as the invalidation message is placed on the bus in a system with a snoopy cache protocol).
- *Invalid,* is data that is no longer up to date. If this data is to be used again, it must first be re-read from main memory (data can also be supplied by another cache if a more recent copy exists there).

The events that give rise to state transitions are summarized in Figure 7.

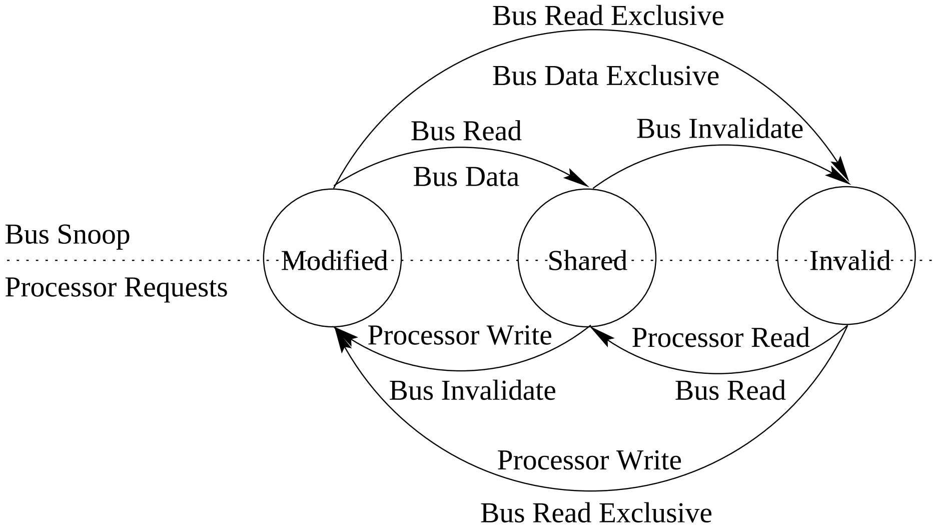

Figure 7. Basic MSI protocol. The requests (above the arrows) are received by the cache controller, and a response (below the arrows) issued by the controller. In the upper part of the figure the requests are received from the bus snooping unit, in the lower part of the figure requests origin from the processor.

2.3.2 Shared Bus

A split-transaction bus scheme with separate address/command bus and data bus can serve two requests at the same time, and allows for more efficient utilization of the bus compared to the classical circuit-switched bus. For instance, a memory read request is split into a request phase, which occupies the address bus during the time it takes the master to send the address to the memory, and a reply phase, which occupies the data bus for the time it takes to transmit the data back to the master. A circuit switched bus would be locked up during the memory lookup, but the split-transaction bus can serve other requests in the meantime. In addition, a fixed number of requests is allowed outstanding at any point in time, allowing buffers to be added, minimizing memory and processor idle times. This mechanism relies on a system where every request gets a tag number. The maximum number of outstanding requests depend on the number of bits in the tag. For example, a system with 4-bit tags can handle 16 outstanding requests. All bus controllers also need to have a request table that keeps track of the outstanding requests (see Figure 8) in order to resolve some special conditions that can arise in split-transaction bus schemes (see [6] for a detailed description of the split-transaction bus).

The bus requests should be served in first-come-first-served order.

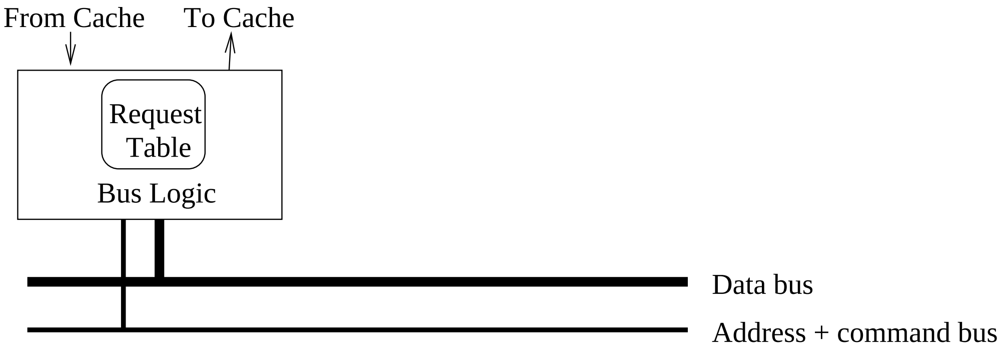

Figure 8. Split transaction bus, the bus logic includes a request table used to keep track of outstanding requests.

2.3.3 Main Memory

The latency and degree of interleaving are the two important variables for the main memory. Multiprocessor systems typically use interleaved memory modules, with buffers for incoming and outgoing requests, in order to increase memory throughput.

2.4 Metrics

Our main concern when evaluating performance is the overall throughput of the system, or how many transactions per unit of time the system can process. From this, the speedup of a multiprocessor over a uniprocessor system can be obtained. The total execution time of a transaction is also of interest. In addition to this, detailed information about processor, bus, memory usage, and queueing times is of importance when trying to identify performance bottlenecks. For instance, long queueing times in the memory system would reveal that memory bandwidth is a bottleneck and degrades the system performance.

We are also interested in how the execution schemes described in Section 2.2 affect performance. Therefore, stall times and time lost because of deadlocks will be measured. This will give a measure of the parallelism in the software. Similar performance issues for load balancing will also be studied by comparing throughput for the different policies.

3 Simulator Design

This section describes how the simulator is constructed. Section 3.1 is an overview and describes the input and inner workings of the simulator, 3.2 explains how the processing units are modeled, 3.3 deals with the memory system, 3.4 with the concept of simulated time, and finally Section 3.5 is about the statistics that are gathered during the simulation, and how they are used.

3.1 Overview of the Simulator

The simulator is built as a number of modules modeling the cache, main memory, interconnection bus, and processing units. A central control unit initiates the model for each simulation run, and acts as a synchronizer during the run (see Figure 9). Each module, once initialized, is self-contained. For every cycle in the simulation, it is enough to tell the module to advance one cycle, and it will know what to do.

The simulator “executes” program code as delays, no actual code execution is performed. Memory references are generated randomly, based on statistical data of read and write instruction frequencies, during the “execution”. This way, memory accesses, cache effects, queueing, and contention can be included in the simulation.

The program was written from scratch. The implementation language is C++, except for the parser which is written in C, with most of the code generated using the GNU parser generator tools bison and flex. The simulator has been successfully compiled and run under Solaris/SPARC, and Linux/x86.

3.1.1 Input

The input to the simulator consists of information from the target software system, and a number of parameters specifying the target architecture that is to be examined.

The information available from the target software system is files containing information captured from a real system. These data files include the functions that were executed during each subtask in a transaction, execution order, the messages sent by these functions, and the execution time for each subtask. The execution times, and the average execution time per instruction from the same system, is used to compute an approximate instruction count for each function.

The data files are parsed, and the extracted information used to build up data structures (i.e. lists) for each transaction. Whenever a new transaction is to be injected into the system by the control unit, one is randomly chosen from the set of available transactions. The processors also use this information to fetch the next function and message information.

Statistical information from cache simulations on a program-driven uniprocessor simulator is the other kind of information from the existing system. This includes the frequency of read and write instructions, average footprint size of the subtasks, and the fraction of the footprint that is instructions and data.

Finally, the parameters that specifies the simulation scenario include number of processors, which execution scheme and scheduling policy to use, cache sizes and cache block size, architecture model (see Figure 6), memory interleaving and latency, and size of queues. See Appendix A for a complete list of simulator parameters.

3.1.2 Internal Communication

All communication between modules are carried out using a request data structure. A request can be any kind of memory/bus transaction, such as a read, a write, or an invalidation. The request data structure contains information about the kind of request, data size, originating transaction number etc. These structures are used for passing information between modules, for queueing of requests, and for run-time statistics gathering.

In addition to the requests, each module have functions used to query and in some cases modify its status. These functions are used to implement the execution schemes and load balancing, and would in a real system need to be replaced by some monitoring software.

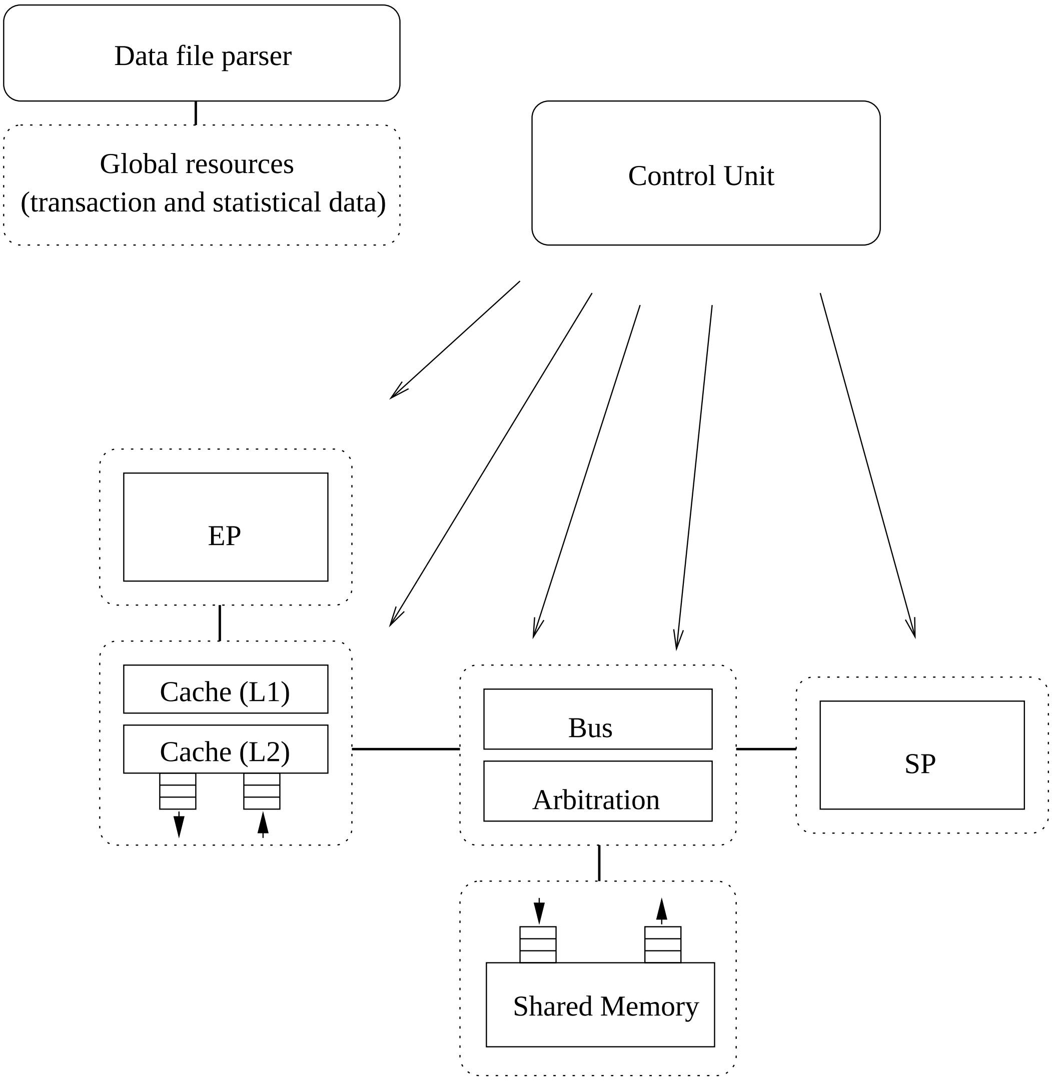

Figure 9. Block diagram of the simulator. Note that there can be any number of EP/ Cache, SP, and memory units connected to the bus.

3.2 Processing Units

There are two types of processing units, an execution processor (EP) and a scheduling processor (SP).

The scheduling processors mainly work as a queue of messages. The messages are distributed among the SPs somewhat differently depending on the load distribution scheme (see Section 2.2.2). In the base architecture, messages are always sent to the SP with the shortest queue of messages, the SP then schedules its message to an EP (always the same one) in FIFO order. The solution to schedule all subtasks to the same EP (transaction scheduling method) is implemented using the architecture depicted in Figure 6B. In this case, a new transaction is scheduled to the SP-EP pair with the lowest load. All new messages produced within this transaction are always sent to the same SP. Finally, the technique with one central SP, see Figure 6D, is somewhat different, since it is not always the case that the oldest message in the FIFO queue is scheduled first. Instead, the SP makes sure that no subtask from the same transaction is still executing before a message can be dispatched. If another subtask in the transaction is still being processed, the SP tries to bypass the message and dispatch the next message in the FIFO queue. In addition, with this approach one SP is queuing and dispatching messages for all the EPs.

The EP "executes" the functions in the subtask triggered by a message. During execution, memory requests are generated according to read and write operation frequency. The number of cycles between memory requests for the EPs is assumed to be uniformly distributed. In real applications, memory references often arrive in "blocks", but this behavior is hard to model, so the simplified assumption of uniform distribution has been made. The EPs keep a history of executed functions as long as it is in the same subtask. When the subtask is finished, the history list is cleared and statistics updated.

The support for the execution schemes can be summarized as follows:

- The ideal scheme needs no special support.
- Execution scheme A is supported by, at the beginning of every new function, poll the other EPs asking for the name of the currently executing functions. If any EP is already using the function, execution is simply suspended until it has finished.
- Execution scheme B is similar to A with the exception that the history lists of the other EPs are searched for the function name, not just the current functions.

As explained in Section 2.2, execution scheme B might end up in a deadlock situation. The history lists are used to implement a deadlock detection mechanism. Each time a processor requests a lock on a function and is made to wait, the processor that currently has the lock is probed, if this processor is also waiting to obtain a lock, the processor that has that lock is probed, and so on. A circular wait is detected if the same EP is visited twice during this procedure. If this happens, the subtask that was started most recently is forced to release all its locks and restart. Figure 10 illustrates the detection mechanism.

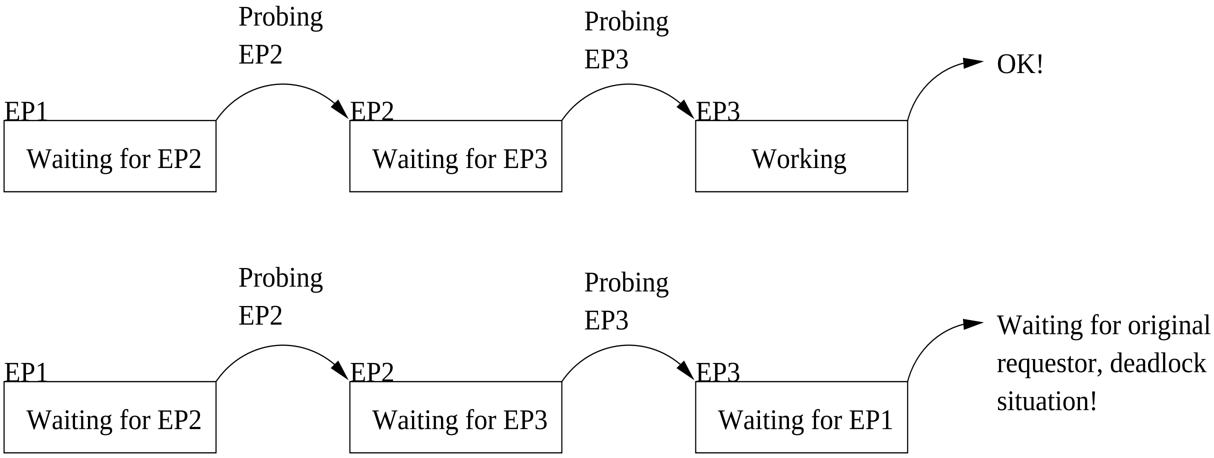

Figure 10. The situation in the upper part of the figure is bad for performance, but will not result in a deadlock. In the figure below, however, a deadlock is detected and must be resolved using rollback on one of the EPs.

Deadlocks are resolved by "magic rollback", which means it is assumed that rollback is possible, but it will not add any time penalty in the simulations, since the length of such a time penalty would be pure speculation anyway. In the simulator, rollback simply means that the subtask is restarted, and all the locks held by the processor are released.

It should be noted that since load distribution, execution scheme, and deadlock detection mechanisms are handled by an "omnipotent monitor" in the simulator, they do not affect the execution of the transactions. In a real system, all of these mechanisms would need some additional processing and likely affect the execution time of the transactions, depending on how they are implemented.

3.3 Memory System

3.3.1 Shared Bus

The split-transaction bus in the simulator is implemented similarly to its corresponding hardware bus. Each transaction is given a tag when put upon the bus, and a table of outstanding requests is maintained. The difference is that the simulator keeps a central request table, while real hardware use a distributed list, one for each bus controller.

If the bus is occupied, or the maximum number of tags is already reached, the request will not be granted bus access. Instead, it will be added to the “arbitration list”. The arbitration list is simply a FIFO used to make sure requests are served in a first-come-first-served order.

When a request enters the bus, the number of cycles required for transport is computed. For memory requests, the data transfer size is always the cache line size, but it may vary for messages. An address bus request always takes the same amount of time. After the request has completed, the next request in the arbitration queue is serviced.

The bus also contains functions for load balancing used when many SPs are connected to the bus. Messages are redirected to the SP with the lowest load (i.e. the shortest message queue). This function should be handled by system software on a real machine.

3.3.2 Main Memory

The memory system serves requests from the input buffer on a first-come-first-served basis. The sizes of the input and output buffers are variable, as is the memory latency.

The degree of interleaving is adjustable (where the number of modules is 2^n, n>0). Since no data regarding locality or distribution of data across modules is available, it is assumed that memory references are uniformly spread out across the modules. This is not entirely true, since there is always some amount of locality in every application, but it is the best we can do at this time. Intuitively, this approximation should work better the more EPs that are used, since memory request from different functions will then be mixed.

The memory requests from the EPs are only addressed to “unknown memory module”, and the specific memory module is not picked out until the request appears on the bus.

3.3.3 Cache Memory

The cache organization described in Section 2.3.1 can give rise to a number of cache events. First, the events that can successfully be served by the cache with no bus traffic are read hits or write hits to data in modified state. We can classify these as one of Level one (L1) cache hit or L2 hit, where the L2 cache has a larger latency than the L1 cache. In addition, there are three cases which will result in bus activity:

- *Read-miss* or *write-miss*, the data requested by the processor is not in the cache. A memory read request must be issued to the shared bus.
- *Invalidation*, a write to data in shared state will have to be accompanied by an invalidation broadcast over the shared bus, so that all other copies of the cache line are in state invalid.
- *Writeback*, there are no unused cache lines. If a cache line in a modified state is to be reused, the old contents must first be written back to main memory.

Since no information about addresses exists, the cache behavior has to be approximated in some way. To do this, the footprint[2] of instructions and data has been measured for a large number of tasks. The average footprint size, and the function length, have then been used to assign approximate footprint sizes to all program functions. The cache is then implemented as a FIFO queue. Whenever a new function is run, a

[2] The footprint in this context is the number of different cache blocks accessed during the execution of the function.

new entry in the queue is created. During execution, it will be “filled” as new data and instructions are requested. In this case fill means that the number of cache blocks in shared, modified and invalid states are counted. Instead of using addresses to know when to read a new cacheline, issue invalidations etc. everything is randomized with probabilities calculated from the information in these cache entries. This way, several important properties of the cache can be accounted for.

- *Writeback.* When the cache is full, and a cache block must be cleared, there is a chance that the victim block is modified and must be written back to memory before it is reused. A block from the oldest entry in the cache queue is chosen. The probability of a write back is the fraction of modified blocks in the chosen entry. If the entry becomes empty after a block is removed, the entry is also removed.
- *Cache affinity.* There might be data left in the cache from a previous execution when a function is run. This can be detected since the function name is attached to each entry in the cache FIFO. If old blocks are found, they are pooled with the now ones when used in the calculations described above, that is, blocks that are still in the cache from an earlier invocation of the function can be used again without a new read.
- *Invalidations.* The frequency of invalidations can be approximated by using information about the contents of all other caches in the system, looking for a matching function name. If one is found, the probability that it contains the same data is dependent on the amount of shared data in each cache, relative to the amount of data belonging to the function.

As an alternative, the simulator supports the use of fixed probabilities. The information required is L1 and L2 hit rates, the frequency of write back, and the frequency of invalidation actions. What you loose with this simplified approach is any kind of measure of cache affinity and warm-up for every new function, and how write backs and invalidations vary with different numbers of processors. This approach has not been used in the case study.

Figure 11 shows how cache events are related.

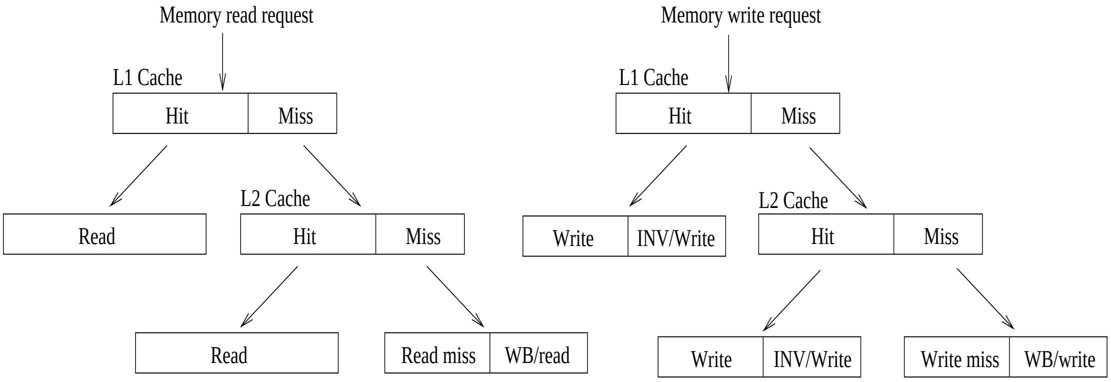

Figure 11. Flowchart for identifying cache events. (WB - Write Back, INV - invalidation message) On a read or write miss, the memory request must use the shared bus. WB and INV events also use the bus.

3.4 System Time

The control unit of the simulator keeps the central “clock” and explicitly tells each unit to advance time by one cycle. This way, it is also possible to run the processors and bus at different clock speeds, which is the common case in modern systems. Each module does, however, keep track of its own activities and knows what to do next.

3.5 Performance Metrics

The simulator collects statistics during run-time, this gives us information about different aspects of the system performance.

The EP time is split up according to its state. After a simulation run, the fraction of time spent in each state is presented:

- *Work* - or computation time, is the time spent executing instructions.
- *Wait* - or stall time, is when the EP is waiting for a memory request. This time is further subdivided in the specific request types; data read, data write, instruction read and handling messages from the SP.
- *Idle* - the time during which no process was scheduled on the EP.
- *Blocked* - time spent waiting to get a lock on a critical region.

In addition, some tests have been performed on code that might end up in deadlock. In these cases, one of the processes are restarted in order to solve the deadlock situation. The simulator also sums “wasted time”. Wasted time is time spent executing on a subtask which is restarted as part of a rollback, and thus the results of the execution are never used. The number of deadlocks that occurred are also counted. As mentioned, rollback is assumed to have no cost, which is not realistic. However, by looking at the number of deadlocks, one can at least get a feeling for how big an impact this problem can be expected to have.

For the data bus and address bus, idle time and working time are accounted for to measure bus utilization. In addition, queueing time[3] is given as a fraction of working time, giving a measure of how much queueing affects the total transfer time. For instance, if queuing time exceeds 1.0, the average request spends more time waiting for the bus than using the bus. The working time is subdivided in time spent on each type of request; read, write, invalidate, write back, and time spent transferring messages between the EPs and SPs.

The utilization of the main memory is measured in the same way, with working, idle, and queueing times for each memory module.

Lastly, there are some overall performance figures (total for all processors measured as the averages during the entire simulation time):

- Throughput (transactions/s) and (transaction subtasks/s)
- Average transaction working time (ns) and average subtask working time (ns).

3.6 Model Validation and Model Accuracy

The best way to validate that the model is correct would be to tune in the parameters as close as possible to an existing system and compare simulated values against values measured on that system. However, the time and resources has not allowed a detailed analysis of the correctness of the simulator. In addition, there are no existing multiprocessor systems to compare with, so the only candidate would be a uniprocessor, which is quite different than the simulated system. Therefore, the validation in this study is restricted to “sanity checks” of the statistics and results produced.

The method used simplifies real-world use by not executing any real code. Memory references are randomly generated, using a uniform distribution, based on measured frequency of reads and writes. This approximation decreases the accuracy of the simulator. Another limitation is the fact that the application is assumed to be the only software to run on the system. In real life, other software, such as an operating system, interpreter, and transaction scheduler would also compete for computation time.

Because of these limitations of the distribution-driven approach, and the lack of any accurate verification, the performance measurements should not be interpreted as absolute answers. Instead, the important conclusions comes from comparisons of different methods and hardware choices, and the relative differences between those methods. These limitations should be kept in mind when interpreting the results.

[3] Queuing time for the bus is the time a request spends in the output queue of a cache (or memory) before it is given access to the bus.

4 Case Study

This case study is meant to cover some scalability issues, and discover the performance bottlenecks as the number of processors increase.

For most of the tests, i.e. if not otherwise stated, the base system is configured as follows: The processor speed is 500MHz, and the bus speed one fifth of the processor speed. The bus width is 256 bit, the L1 caches (data and instruction) are 64kB each, and the L2 cache is 1MB. The main memory has 60ns latency, and is 2-way interleaved. A cache line is 64 bytes. The program code contains 20% read, and 10% write instructions. All measurements are made on a system under high load. Execution Scheme A and Lazy scheduling are used as default settings in all tests where they are not explicitly mentioned.

The hardware parameters are loosely based on those of a recent AlphaServer system[4], but a direct comparison is not possible. For instance, the EPs of the simulator always execute one instruction per cycle, while the superscalar Alpha 21264 often executes more than one instruction per cycle on average, depending on the application.

The choice of a 1MB L2 cache can seem conservative, considering that the current Alpha 21264 processors support up to 16MB. The motivation is that the simulations only use a subset of the applications available on the examined system. A too large cache could mean all or most of the instructions and data would fit into the cache, which would yield misleading results. Therefore a more conservative choice was made.

The architecture used as the base case is the system depicted in Figure 6A.

4.1 Software Related Issues

As described in Section 2.2.1, the amount of parallelism depend on the software, and two techniques for solving data inconsistency problems under different circumstances, *execution scheme A and B,* were introduced. Section 4.1.1 explores how these two schemes perform, compared to an ideal case where no data inconsistency problems exists. The different approaches to load balancing described in Section 2.2.2 are examined in 4.1.2.

4.1.1 Parallelism

Figure 12 shows the throughput for each of the execution schemes, plotted against the ideal scheme for 1, 2, 4, and 8 execution processors.

Clearly, the use of execution scheme A limits the parallelism somewhat, compared to the ideal case. This is expected, since execution is stalled when the EP is waiting to get a lock on a function. Figure 13 reveals that the blocking time is only about 1.7% of the cycles for 2 processors, rising to 3.5% for four processors, and 8.3% for 8 processors.

For execution scheme B, blocking time is much worse that for scheme A. 7%, 24%, and 45% for 2, 4, and 8 processors respectively. In addition, some of the execution time is wasted because of rollbacks caused by deadlock situations.

For 2 processors, there are on average 450 deadlocks per second, or 225 per processor. For 4 processors, that figure rises to 670, and for 8 processors there are 1113 deadlocks/s per processor. The time wasted because functions need to be restarted is still quite low, only about 3% for 8 processors. However, it is important to remember that this figure does not include any penalty for the rollback operation, so the execution time could be significantly larger if the rollback operation is computationally expensive.

Figure 13 shows how the processor time is spent. The numbers are averages over all processors for the multiprocessor configurations. Computing time is the fraction of the execution time that the processor is actually executing code. Data write, data read, and instruction read is the time the EP is stalled because data is not yet available (cache misses). Blocked time is the time the EP is waiting for a lock on a function,

[4] See *http://www.digital.com/alphaserver/alphasrv8400/8x00_tech_summ.html* for a technical summary of high-end AlphaServer systems.

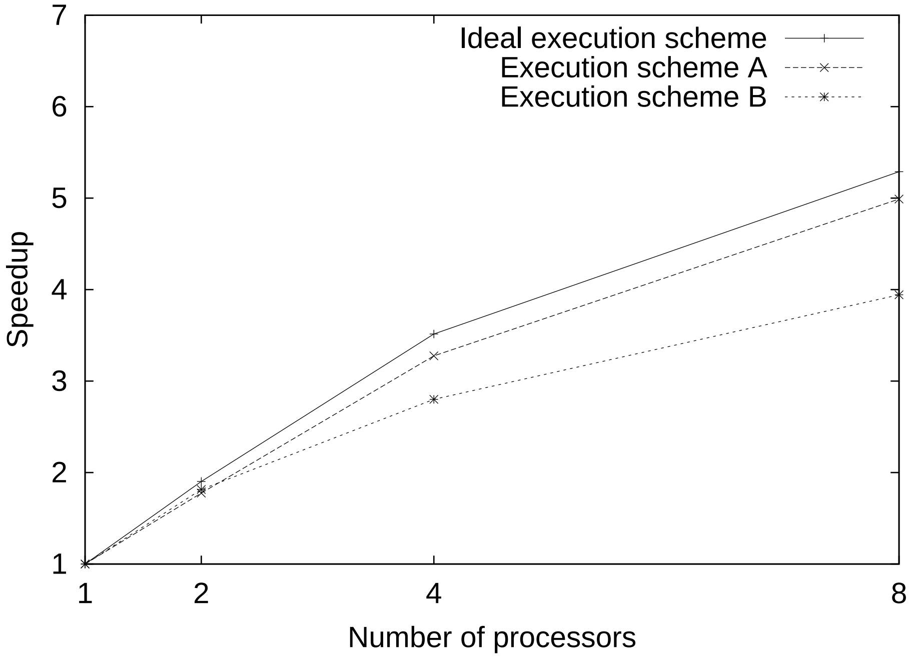

Figure 12. Throughput for different execution schemes on 1, 2, 4, and 8 processors.

this time can not be used for any work. Table 1 shows detailed statistics from a simulation of a four processor system.

Blocking does not degrade performance as much as one would expect, looking at the blocking time in Figure 13. The reason seems to be that queuing time due to bus contention is shorter while one of the processors is blocked (see Table 1), and therefore the other processors can work more efficiently. In fact, the fraction computation time is only a few percent lower than the ideal scheme even when blocking time is 20-40% of the cycles. For instance, the total throughput for execution scheme B is 24% lower that the ideal scheme, although the blocking time is 45%, in the 8 processor simulations.

The important conclusions from this experiment is that the blocking time is growing quickly as processors are added, especially in execution scheme B. The blocking time, together with the time that will need to be used for rollbacks, will be a severe limiting factor on the number of processors that can be used, unless improvements is made on this scheme.

Table 1. Timing information for a four processor system.

Execution Scheme	Speedup over 1 EP system	Computation time [fraction of cycles]	Stall time [fraction of cycles]	Blocked time [fraction of cycles]	Deadlocks [#/s per EP]	Wasted time [fraction of total time]
Ideal	3.514	0.361	0.631	0	0	0
Scheme A	3.276	0.339	0.594	0.059	0	0
Scheme B	2.801	0.294	0.462	0.238	670	0.014

4.1.2 Load Distribution

The scheduling schemes are compared to a base case, and the resulting speedups presented in Figure 14. The scheduling used in the base case is the lazy scheduling, where no enforcement of execution

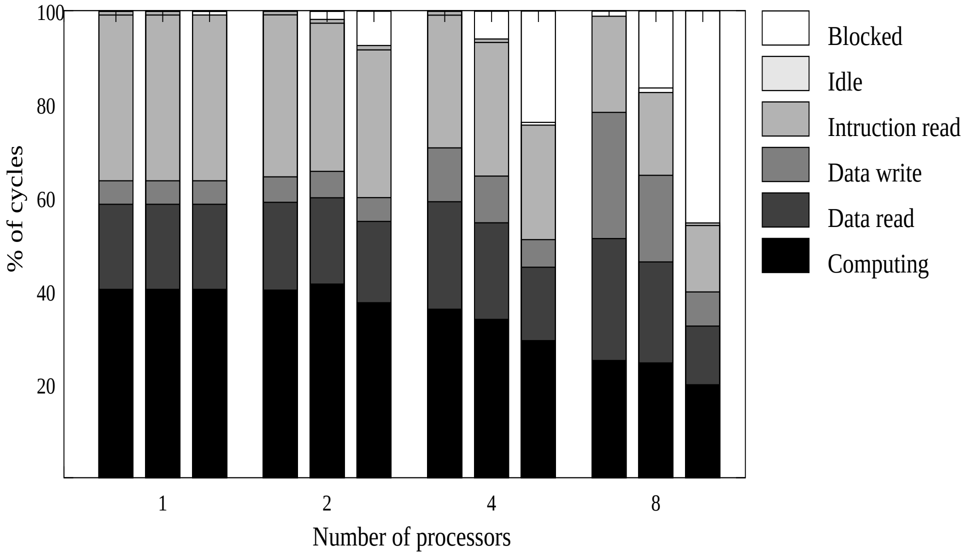

Figure 13. The left column for each number of processors show clock cycle usage for the ideal execution scheme, the middle columns for execution scheme A, and the right column for execution scheme B.

sequence is made. As mentioned in Section 2.2.2, two possible solutions resulting in sequential execution of transactions have been considered.

The solution with local SPs, used for the transaction scheduling method, is faster than the lazy scheduling over the bus. That means the restriction of not being able to run a transaction on several processors is not very severe. In fact, the gain from removing message transfers from the bus more than compensates for that loss. The cache miss rate does not seem to be affected significantly by the choice of method, suggesting that cache affinity is pretty much the same regardless of the method used.

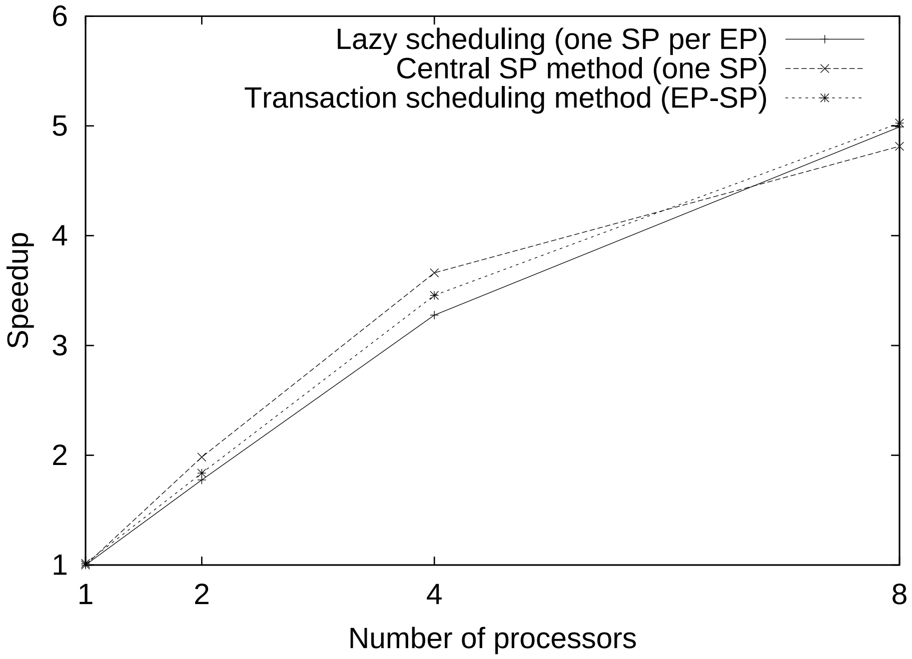

Figure 14. Scheduling approaches. The three scheduling methods compared. The transaction scheduling methods have EPs directly connected to the SPs, while the other methods have the SPs connected to the bus.

The central SP method, where one SP handles scheduling for all EPs, seems to be faster than lazy scheduling, for two or four processors. This is counter-intuitive, since the lazy scheduling does not have the restriction that the subtasks in a transaction must be executed sequentially. We believe this might be because the implementation of the lazy scheduling, where messages always sent to the SP with shortest queue, might not be the best. The single SP method, on the other hand, does the reverse and looks at which EPs are idle when deciding where a message should go. This issue will be investigated further in the future.

This experiment tells us that sequential scheduling of subtasks in a transaction is not a severe limitation. On the contrary, the tightly coupled EP-SP in the transaction scheduling method leads to performance gains.

4.2 Hardware Related Issues

Next, a few hardware related parameters will be examined. First, in Section 4.2.1, the architectures in Figure 6 B and C are compared to the base architecture, shown in Figure 6A. In 4.2.2 the impact of bus width is examined, and finally in 4.2.3, we look at memory interleaving.

4.2.1 Architecture

As Figure 15 shows, connecting the SPs directly to the EPs gives a small performance advantage over connecting them to the shared bus, despite the fact that the scheduling is more restrictive (see Section 4.1.2). The performance gain can be attributed to shorter latency for message communications, since messages go directly from the SP to the EP instead of over the bus. In addition, the removal of messages from the bus means less bus traffic, about 10% for an 8 processor system, which shortens the bus queuing times for the remaining requests.

The architecture with local memory gives a performance advantage for systems with more than two processors. This can be explained because the local memory accesses are faster, since they do not need the shared bus, and there will not be any bus contention for those accesses. The values are obtained under the assumption that there are local copies of the instructions, or the program code, in all nodes, while the data is spread out evenly among the nodes. The shared bus is used when data is located in a remote memory, and for coherency traffic. The local memories are 2-way interleaved in the simulations.

Table 2 shows that stall times due to reads and writes are lower for both alternative architectures, compared to the base system, which is a result of the lower bus traffic. Consequently, the computation time is higher.

Both these alternative (compared to the base case) methods does give a slight performance advantage. The local SP approach has the additional benefit of solving the sequential scheduling problem.

Table 2. Timing information for a four processor system.

Architecture	Speedup over 1 EP system	Computation time [fraction of cycles]	Stall time [fraction of cycles]	Blocked time [fraction of cycles]
SPs connected to bus	3.276	0.200	0.738	0.052
SPs connected to EPs	3.456	0.209	0.669	0.045
Local memory modules	3.520	0.229	0.649	0.046

4.2.2 Shared Bus

Figure 16 shows throughput for various bus widths. Using only one processor, performance is not significantly affected by the bus width, since bus utilization in very low. Using 4 or 8 processors, a bus width of less than 128 bits have a larger impact. For an 8 processor system, the throughput is 30% lower with a 32-bit bus than a 256-bit bus.

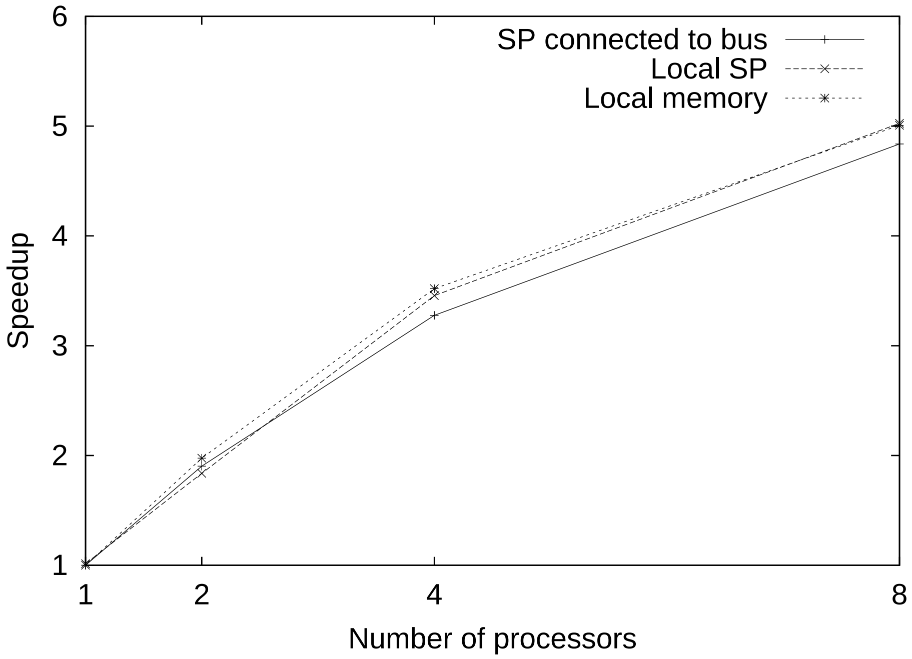

Figure 15. Comparison of placement of the SPs, directly connected to each CP, or via shared bus, and use of local memory modules.

The reason is, obviously, that transfer times are higher for a narrower bus, which also increases queuing times. Table 3 shows queuing times for a number of different systems with 1-8 processors, and bus widths ranging from 32-256 bits.

The bus queuing times are found to be quite sensitive to many parameters, for instance the fraction of read and writes in the software, and the footprint (larger footprint gives lower cache hit rates). For the values used in Section Table 3 and Section Figure 16, a 128-bit bus seems sufficient for the four and eight processor systems, and a 64-bit bus for 2 processors. But it is obvious that queuing times rise rapidly if the bus width is not sufficient, so caution should be taken.

Table 3. Average queuing times, in bus controllers, for bus requests. Queuing times are measures as fraction of bus working time, that is for a value larger than one, a request spends more time waiting for the bus than using the bus.

Number of EPs	Bus Queuing time for 32 bit wide bus	Bus Queuing time for 64 bit wide bus	Bus Queuing time for 128 bit wide bus	Bus Queuing time for 256 bit wide bus
1	1.740	1.591	1.502	1.472
2	1.136	1.043	1.002	1.023
4	3.134	1.726	1.701	1.969
8	7.227	5.849	5.553	3.184

4.2.3 Main Memory

As revealed by Figure 17, memory interleaving does not affect performance to a significant degree for one or two EPs. For 4 and 8 EPs however, a 2-way or 4-way interleaved memory does give noticeable performance gains. For a 4 EP system, 4-way interleaved memory gives a 12% performance gain over a system with a single memory module. Table 4 show average queuing times for memory accesses.

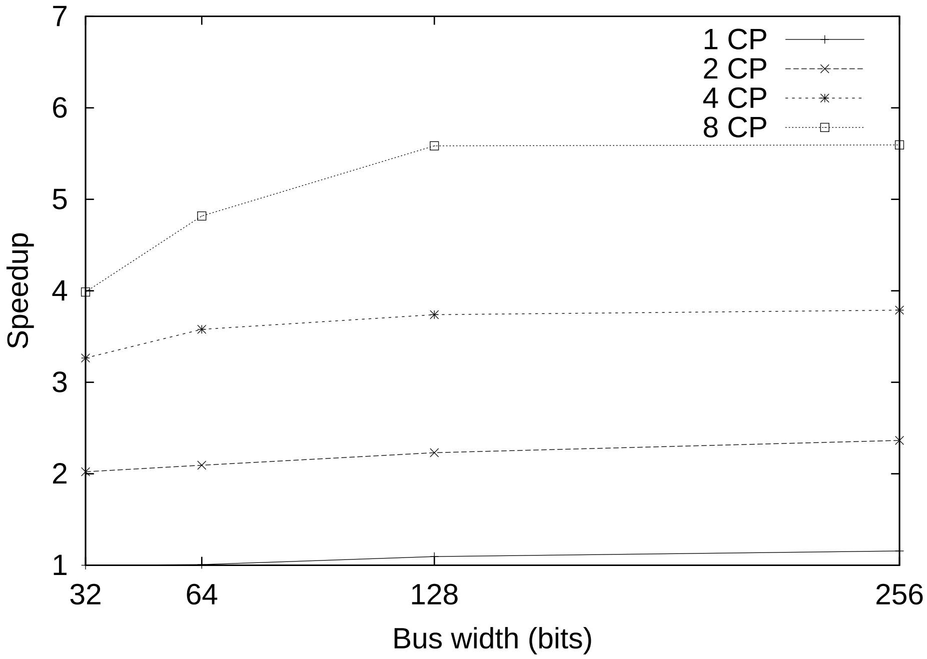

Figure 16. Effect of bus width on systems with 1, 2, 4, and 8 processors.

Table 4. Average queuing times, in the memory modules, for memory requests. The queuing time is related to the lookup time, for instance, a queuing time of 0.5 means the total time for a memory lookup is 50% longer than the actual memory lookup time. Note: For 1 EP all queuing times are under 0.01.

Number of EPs	Memory Queuing time for a single memory module.	Memory Queuing time for 2-way interleaved memory.	Memory Queuing time for 4-way interleaved memory.	Memory Queuing time for 8-way interleaved memory.
2	0.03079	0.02619	0.01134	0.00508
4	0.25009	0.04795	0.04123	0.02162
8	0.47571	0.08965	0.05693	0.01947

In general, the interleaving should be scaled with the number of processors, in order to minimize memory access delays. At least for an eight processor system, the gain of 8-way interleaving over 1-way is large enough that interleaving makes sense.

In this section, the simulation platform has been used in a case study. The results for this particular system show that the key considerations are software parallelisation, and bus bandwidth. The other issues investigated do also show noticeable differences in performance, however. The most striking performance limitation found was the increase in blocking time as the number of processors increase for execution scheme B. The speedup shows 24% lower performance for that scheme than the ideal execution scheme in an eight processor system.

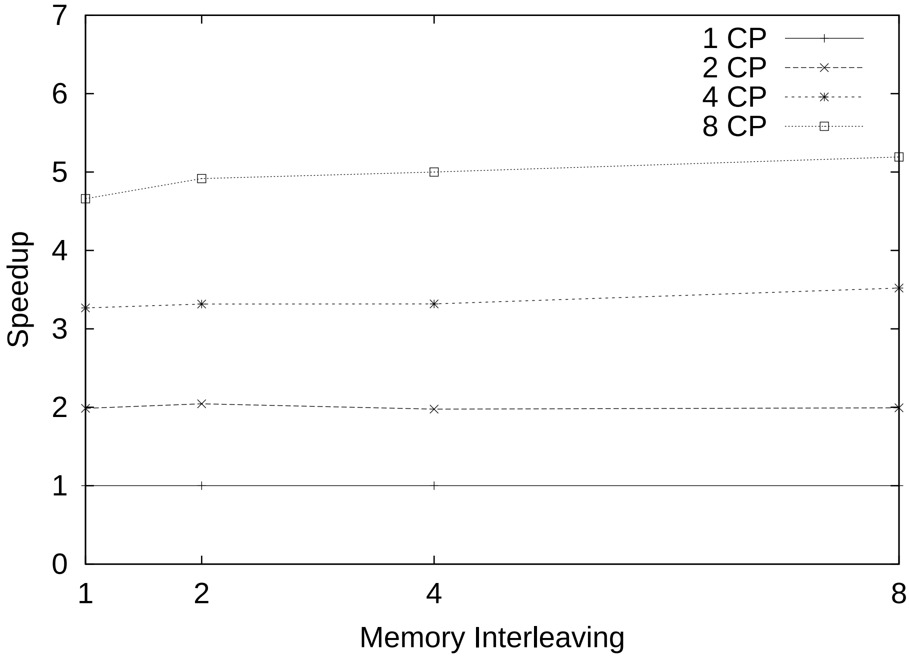

Figure 17. Memory interleaving.

5 Conclusions

A simulator with distribution-driven workload has been designed and implemented in order to evaluate the performance of a throughput-oriented telecom system in various multiprocessor configurations, and with different assumptions regarding existing software.

The simulator uses data files with information about the software, including which functions are executed during a transaction, execution times, and messages sent by each function, as well as statistics on read and write frequency in the software, and the footprint of each function. This information is used to recreate the conditions in the system under examination as closely as possible. Thanks to this, characteristics such as cache misses, coherency traffic, cache affinity, and bus contention are taken into account by the simulator. In addition, some software issues, such as parallelisation and load balancing, can be investigated. The simulator that has been implemented is fairly flexible, and the implementation time short due to the choice of a distribution-driven method. The trade-off, compared to instruction-level simulation, is lower accuracy.

In the case study, it was discovered that when functions in a part of a transaction, a subtask, is locked until the subtask is completed in order to enforce mutual exclusion on functions, the parallelism is limited to a significant degree. This becomes increasingly serious as the number of processors is increased. For 8 processors, the performance drops 24 percent below the ideal execution scheme, but it still does gives a significant speedup compared to a uniprocessor system. Still, this will be a bottleneck for larger systems, so it would be worthwhile to work on improvements on this scheme.

The placement of the scheduling processor (SP) directly connected to the execution processor (EP), rather than connected to the bus, seems to make sense, even if one does not take the sequence of execution into account. The greater flexibility you gain by having the SPs on the bus, and being able to schedule individual subtasks onto any EP, does not compensate for the longer execution times this method yields, at least not for systems with four or more processors. But the difference is not that great, both methods are viable options.

Overall, parallelisation and adequate bus width are the most important factors investigated in the case study. These two are also the most sensitive when adding more processors. Scheduling policy, architecture, and degree of memory interleaving have a lower impact, but could still yield some performance gain if the the right choices are made.

References

[1] M. Azimi, and C. Erickson, “A software Approach to Multiprocessor Trace Generation”, in *Proceedings of the Computer Software and Applications Conference*, pp. 99-105, 1990.

[2] R. C. Bedichek, “Talisman: Fast and Accurate Multicomputer Simulation”, in *Proceedings of the ‘95 ACM SIGMETRICS Conference*, pp. 14-24, 1995.

[3] J. B. Chen and B. N. Bershad, “The impact of operating system performance on memory system performance”, in *Proceedings of the 14th ACM symposium on Operating System Principles*, pp. 120-133, December 1993.

[4] M.-C. Chiang and G. S. Sohi, “Evaluating Design Choices for Shared Bus Multiprocessors in a Throughput-Oriented Environment”, *IEEE Transactions on Computers*, pp.297-317, Vol. 41. No. 3. March 1992.

[5] R. G. Covington, S. Dwarkadas, J. R. Jump, S. Madala, and J. B. Sinclair, “The Efficient Simulation of Parallel Computer Systems”, *International Journal in Computer Simulation*, pp. 31-58, Vol. 1. January 1991.

[6] D. E. Culler, J. P. Singh with A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, San Francisco, CA, Morgan Kaufmann Publishers, 1999.

[7] S. J. Eggers and R. H. Katz, “A Characterization of sharing in Parallel Programs”, in *Proceedings of 15th Annual Symposium Computers Architecture*, pp. 373-382, June 1988.

[8] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, 1993.

[9] C.-S. Lee, and T.-M. Parng, “A subsystem-Oriented Performance Analysis Methodology for Shared-Bus Multiprocessors”, in *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7. No. 7. July 1996.

[10] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner, “SimICS/sun4m: A Virtual Workstation”, in *Proceedings of Usenix Annual Technical Conference*, June 1998.

[11] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, “Complete Computer Simulation: The SimOS Approach”, in *IEEE Parallel and Distributed Technology*, pp. 34-43, 1995.

[12] Richard A. Uhlig and Trevor N. Mudge, “Trace-Driven Memory Simulation: A Survey”, in *ACM Computing Surveys*, pp.128-170, Vol. 29. No. 2. June 1997.

[13] A. Silberschatz, and P. B. Galvin, *Operating System Concepts*, Addison-Wesley Publishing Company, 1998.

[14] M. K. Vernon, E. D. Lazowska, and J. Zahorjan, “An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols”, in *Proceedings of 15th Annual Symposium Computers Architecture*, pp. 308-315, June 1988.

Appendix A - Simulator Configuration

The simulator parameters are set using a configuration file. There are parameters for each kind of module (EP, cache, SP, bus, and memory) as well as some general parameters where it is possible to set, for instance, the different execution schemes, and load balancing policies discussed in Section 2.2.

The simulator is run from a shell, with the name of the configuration file and the files with transaction data as command line arguments. The program is non-interactive (requires no manual input), so it is possible to schedule batches of simulations simply by creating a shell script running the simulator repeatedly with different configuration files.

General:

- Simulation length [number of cycles].
- Statistics gathering starting point [cycle number] - used to rid the statistics of values gathered before a steady state is reached.
- Number of transactions on average issued per second.
- Execution scheme [ideal scheme, scheme A, scheme B].
- Load balancing policy.
- Conversion factor [function length in ns to function length in clock cycles].
- Level of debugging information [none, normal, all].

EP:

- Number of processors.
- Processor clock rate [MHz].
- Frequency of memory writes and reads [percent].

Cache:

- Cache mode [fixed or dynamic frequency behavior].
- Cache block size [bytes].
- L1 cache and L2 cache latency [cycles].
- L1 data and L1 instruction hit rates (only used in fixed mode) [percent].
- L2 hit rate (only used in fixed mode) [percent].
- Writeback frequency (only used in fixed mode) [percent].
- Invalidation frequency (only used in fixed mode) [percent].
- Input and Output buffer sizes.
- L1 data, L1 instruction, and L2 cache sizes [bytes].
- Average footprint size [cache blocks].
- Fractions of footprint that are instructions, shared, and private data [percent].
- Fractions of reads and writes that are made to shared and private data [percent].

SP:

- Message buffer size.
- Computation time [cycles] - the time it takes to prepare an incoming message for scheduling, this is only a factor when the SP is empty, otherwise some other message is probably already prepared.

Bus:

- Bus clock rate [submultiple of processor frequency].
- Bus width [bytes].
- Maximum number of outstanding requests.

Memory:

- Memory latency [ns].
- Size of input and output buffers.
- Interleaving - number of memory modules.

Appendix B - Sample Output From the Simulator

This is the output of a simulation. The first few lines just tells that the input files are read. Then, after the simulation is done, some information is printed. The first paragraph (starting with Simulation time) states some important characteristics that was set in the configuration file. The rest is statistics gathered from the various modules in the simulator.

Noticeable in this particular simulation is the low bus utilization, which is due to a high cache hit rate, and a wide bus. The bus width is in bytes, and the frequencies in MHz.

```
< output starts here >--------------------------------------------------------------------------

Reading Configuration File...

Read 12 sequences. Average function length (instructions): 932

Starting simulation...

100 M cycles simulated...

Simulation time [# of cycles]: 100000000
IPU's: 2  Frequency: 500
Mem modules: 2  Latency: 60 ns
Bus width: 32  Frequency: 100
Execution scheme: B  Load balancing: Lazy

EP UTILIZATION
ep#: 1  Work: 0.355857  Wait: 0.583018  Idle: 0.00668519  Blocked: 0.05444 Wasted: 0.00184893
  Wait time due to -> read: 0.304247  write: 0.0837002  instr. read: 0.609857  messages: 0.00219555
  L1d hitrate: 0.861971  L1i hitrate: 0.916883  L2 hitrate (fraction of L2 lookups): 0.995191
(Fraction of invalidations to cache writes: 0.00341154)
ep#: 2  Work: 0.393574  Wait: 0.502195  Idle: 0.0121087  Blocked: 0.0921214 Wasted: 0.00250514
  Wait time due to -> read: 0.337168  write: 0.103574  instr. read: 0.55638  messages: 0.00287783
  L1d hitrate: 0.897622  L1i hitrate: 0.940916  L2 hitrate (fraction of L2 lookups): 0.995501
(Fraction of invalidations to cache writes: 0.00363488)

Avg.  Work: 0.374716  Wait: 0.542606  Idle: 0.00939695  Blocked: 0.0732807
  Wait time due to -> read: 0.320708  Write: 0.0936373  Instr. read : 0.583118  Messages : 0.00253669
Avg. cumulative work->read->write->iread->idle->blocked: 0.374716 0.548734 0.599542 0.915946 0.925343 0.998624

MEMORY MODULE UTILIZATION
mem# 1  Work: 0.005143  Idle: 0.994857  Mem queing time: 0.0142913
mem# 2  Work: 0.00504133  Idle: 0.994959  Mem queing time: 0.0145993
Avg.  Work: 0.00509217  Idle: 0.994908

ADDRESS BUS UTILIZATION
Idle: 0.98061  Work: 0.0193906 where
   Read      : 0.110208
   Write     : 0.0599547
   Invalidate: 0.155035
   WriteBack : 0.00959803
   Message   : 0.665204
DATA BUS UTILIZATION
  Idle: 0.975687  Work: 0.0243129  where
   Read      : 0.219134
   Write     : 0.119107
   WriteBack : 0.010831
   Message   : 0.650928
Bus queing time: 1.03719

THROUGHPUT(transactions): 408
THROUGHPUT(subtasks): 43776
DEADLOCKS : 81

Average subtask working time (cycles): 4067.9
Average transaction working time (cycles): 5.06202e+06
```