A Computer Architecture Workbench

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Universiteit van Amsterdam,
op gezag van de Rector Magnificus
prof. dr J.J.M. Franse
ten overstaan van een door het College van Promoties ingestelde
commissie in het openbaar te verdedigen in de Aula der Universiteit
op maandag 14 december 1998 te 11.00 uur

door

Andrew David Pimentel

geboren te Haarlem

Promotor:	prof. dr L.O. Hertzberger	(Universiteit van Amsterdam)
Commissie:	prof. dr ir H.J. Sips	(Technische Universiteit Delft)
	prof. dr P.M.A. Sloot	(Universiteit van Amsterdam)
	prof. drs M. Boasson	(Universiteit van Amsterdam)
	dr H.L. Muller	(University of Bristol)
	dr W.G. Vree	(Universiteit van Amsterdam)
Faculteit:	Wiskunde, Informatica, Natuurkunde en Sterrenkunde Kruislaan 403 1098 SJ Amsterdam The Netherlands	

Advanced School for Computing and Imaging

The work that is described in this thesis has been carried out in the ASCI graduate school and was financially supported by:

- Universiteit van Amsterdam, sectie Computer Architectuur & Parallelle Systemen
- Philips Research Laboratories Eindhoven
- Stichting High Performance Computing and Networking

ISBN 90-5776-017-7
ASCI dissertation series number 40.

Cover design: Willemijn de Lint.
Printed at Print Partners Ipskamp, Enschede, The Netherlands.

Acknowledgements

This dissertation could never have been completed without the help, sympathy and participation of many people. Therefore, the people I mention below deserve a round of applause.

First of all, I would like to thank my promoter, Bob Hertzberger. It is he who has guided me during the past five years of research, while giving me a lot of freedom to define the scope of my research. His feedback on drafts of this thesis certainly helped to improve its quality. Bob, I am grateful for your support and the confidence you have placed in me. I really hope that our co-operation will continue to be this fecund in the years to come!

After doing my Master's, I was called up for military service and I suddenly realised that it would be neat to become a conscientious objector (the simultaneous occurrence of these two events was, of course, pure coincidence). Being a conscientious objector, this allowed me to avoid the uselessness of serving in the army and to do something more serious by performing research ... I mean, civil service ... at the IC^{3}A (Interdisciplinary Center for Complex Computer facilities Amsterdam). For this I thank Peter Sloot who arranged my position at the IC^{3}A.

Henk Muller has always been a mentor to me. Literally, he was one of the two graduation mentors for my Master's (Benno Overeinder was the other one). After Henk left to work at the University of Bristol, he continued to show interest in my work. Being one of Henk's apprentice wizards, it should be no surprise that many of the ideas presented in this dissertation are in one way or another influenced by his previous and current work. Henk, I hope to continue learning from you in the future. I thank you for giving me excellent comments on draft versions of my thesis.

Special thanks go to two of my direct colleagues, namely Marcel "ace" Beemster and Jon "type-checked" Mountjoy. During the past five years, Marcel has been a continuous source of knowledge to me. Marcel, I am grateful for your patience to answer all my questions and for allowing me to constantly borrow your books. You also did an excellent job in producing valuable comments on this thesis. Thanks and keep on flying!

Jon is a person of the right type (I type-checked this). He has always been willing to help me whenever my knowledge on the English language or on LaTeX matters let me down. Jon, I really appreciate your help and I will stop teasing you with your love for lazy (dis-) functional languages ;-)

John van Brummen and Henk Kok directly contributed to this dissertation. John developed the initial version of Mermaid's reality-based workload modelling framework and Henk constructed Mermaid's GUI for run-time visualisation. Guys, you did a great job, thanks!

From Philips Research Laboratories in Eindhoven, I would like to thank all members of the PROMMPT project team. Special thanks go to (in alphabetical order) Rudi Bloks, Jos van Eijndhoven, Evert Jan Pol, Frans Sijstermans, Pieter Struik, Marcel Tromp, Kees Vissers and Pieter v/d Wolf for their direct involvement in my work.

Jonas Karlsson introduced me to the secrets of LH*LH. We have had several stimulating conversations on the implementation details of this data structure. Jonas, thanks for your help.

I would like to thank all the other (ex-)colleagues I have worked or socialised with. Thanks Alfons, Arjan, Arjen, Ben, Benno, Berry, Dick, Drona, Edwin, Frank, Hamideh, Hugh, Jaap, Jan, Jeroen, Joep, Martin, Pieter, Robert, Ruud, Theodossis, Theun, Toto and Wim. Our secretariat deserves much gratitude for helping me with all kinds of organisational matters. They have made my life a whole lot easier. Thanks Hugo, Ina (who unfortunately is not with us anymore), Laura, Monique and Virginie! I also thank Jan, Gert and Robbert from our system group for their support and help.

There are, of course, people who were not directly involved in my work at the university but who still earn a lot of appreciation. I thank my parents for everything they taught me. Now, I am reaping the fruits of their lessons.

Last but certainly not least, I thank my dear wife Caroline for having such patience with me. Never did she complain about the evenings, and sometimes nights, I spent behind my computer or reading papers.

11th November 1998

Andy Pimentel

Contents

Chapter 1

Introduction

"Where a calculator on the ENIAC is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1,000 vacuum tubes and perhaps weigh 1-1/2 tons."

Popular Mechanics, 1949

The design of computer architectures is a difficult task due to the many tradeoffs and the large design space architects have to deal with. Although good tools exist to aid the computer architect in the design-phase, there are no real formalisms according to which the actual design of the architecture should take place. Hence, much is left to the architect's experience and intuition. It is for this reason that computer architecture design is often called an *art* rather than a science.

Most computer architects apply a top-down approach and, for this reason, use a design cycle which is similar to the one shown in Figure 1.1. First, an initial architecture design is made and specified at a low level of detail. Here, the architect should take the requirements of the target architecture (e.g. the type of processor, I/O bandwidth, etc.) and the target applications (e.g. the programming model) into consideration. Using step-wise refinement, the design is refined until it is specified at a level at which it can be implemented. This approach of step-wise refinement is an iterative process in which the architect is guided by feedback obtained by evaluating (parts of) the intermediate designs. Typically, this evaluation aims at studying four characteristics of the architecture [97]: functionality, performance, cost and physical requirements.

When evaluating the functionality of a design, it is checked whether or not the proposed architecture works as it should, i.e. its functional correctness is verified. For this purpose, quite a few tools are available of which many perform a detailed simulation of the architecture. In other words, these tools *imitate* the functionality of the architecture in software. To verify the architecture design, the simulation output for user-supplied input values is compared to the expected values. This inferential approach carries, however, the risk that some aspect of the design may not be tested. Therefore, an alternative approach based on formal methods is gaining popularity. These formal verification tools use deductive reasoning techniques borrowed from mathematics to compare the logic of an architecture directly against the logic expressed in a functional specification. Although these techniques are promising,

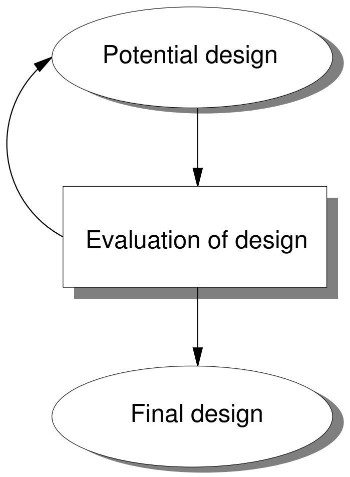


Figure 1.1: A typical design cycle.

they are still in their infancy and have difficulties with the verification of complex computer architecture designs. A comprehensive discussion on formal verification techniques can be found in [139].

To gain insight into the attainable performance of an architecture design and to detect potential bottlenecks, a performance model of the architecture can be built. Doing so, one could, for example, check whether or not the designed architecture meets the specified (if any) real-time requirements. Moreover, the modelling of architecture performance also allows for the scenario analysis necessary for studying the consequences of design decisions. In other words, the performance model can give feedback to the architect regarding *what-if* questions. It could, for example, give an estimation of the performance impact when changing (e.g. enlarging) the data cache of the processor.

In the past few decades, much research effort has been put in the performance evaluation of computer architectures and, as a consequence, many techniques (and tools) exist. Generally, all of these techniques are based on either *analytical modelling* or modelling by means of *simulation*. In case of the first, a performance model is built using mathematical equations [81, 59]. The main advantage of analytical modelling is its speed (response time). When changing a parameter value in the model (e.g. the number of processing elements in the system), a new performance estimation can be made in an instance. This allows the architect to evaluate a large number of architecture designs, even the ones with ridiculously large system parameters (e.g. 1 million processing elements).

Another pleasant feature of analytic models is that there exists a clear relation between the model parameters. On the other hand, the most important drawback of this performance evaluation technique is that accurate analytic models may easily become very complex. In particular, the dynamic behaviour within architectures, such as contention in a network, is hard to model accurately. For a more elaborate discussion on analytical modelling, the interested reader is referred to [1], which presents a good description of an analytic cache model. Alternatively, in van Gemund's dissertation [140], an excellent overview is given of the differences between several analytical modelling techniques when evaluating the performance of parallel systems.

In contrast to analytical methods, modelling by means of simulation can easily keep track of the dynamics in architecture behaviour. In fact, simulation models are capable of modelling architectures at any required level of detail or, to put it differently, at any degree of accuracy. There is, however, a tradeoff between simulation accuracy and efficiency: the lower the level of detail of the simulator, the higher the computational demands of the simulation. Therefore, the major drawback of simulation is that it can be very computationally demanding.

Generally, analytical modelling and simulation are supplementary rather than competitive evaluation techniques. Although analytical modelling is more restricted than modelling by means of simulation (it has a smaller domain of applicability), it often provides insight as to how a model behaves under *arbitrary* experimental conditions. By contrast, one simulation run only provides information on how the model behaves under the set of experimental conditions applied during the simulation run. So, to quote François Cellier [20]:

> “Only an idiot uses simulation *in place* of analytical techniques”

Compared to the rigorous methods to evaluate the functionality and performance of architectures, the tools to evaluate the architecture’s cost and physical requirements, such as the size or the power dissipation, are still in their infancy. As a consequence, this area of architecture evaluation still heavily depends on the architect’s expertise.

It is, of course, dependent on the nature of the target architecture which of the four discussed evaluation aspects is important and which one is not. For example, a designer of processors for calculators is more interested in the physical requirements and cost of the design than in the performance. On the other hand, a supercomputer architect focuses on performance rather than on cost and physical requirements.

In this thesis, we address the design of high-performance parallel computer architectures and, in particular, we focus on simulation techniques for the performance evaluation of these computer architectures. Before going into the details of computer architecture simulation, the remainder of this chapter gives a short introduction on parallel computing and on the computer architectures commonly used in this field.

1.1 Parallel computing

The development of faster computers is an ever continuing goal for computer architects. Since the introduction of the Intel 4004 microprocessor in 1971, for instance, the performance of microprocessors has increased with more than three orders of magnitude. This significant achievement is due to the many architectural and physical improvements that have been made to microprocessors in the last quarter of this century. Instruction pipelining, caching and branch prediction are just a few examples of the techniques, which were already known from the field of supercomputer and mainframe architecture, that have been introduced in microprocessor architectures to yield higher performance. Additionally, the clock frequencies of microprocessors have increased from under the 1 MHz (the Intel 4004) to several hundreds of MHz (e.g. the 600 MHz DEC Alpha 21164). In fact, 1 GHz (and beyond) microprocessors are currently being announced. Moreover, the rapid advances in packaging technology allow for higher transistor densities, which makes it possible to increase the functionality (and thus more processing power or bigger caches) on a chip.

Unfortunately, there are some fundamental limitations that form, or will form, an obstacle to the microprocessor performance push. First, the transmission speed on a chip is limited by the speed of light. With the high clock frequencies and the small feature size of chips, the large wire delays become an increasingly important dilemma which processor architects have to face [14, 102, 26]. Second, the reduction of the size of chip components is also constrained by physical limitations. As a matter of fact, efforts are already being made to explore this limit by assembling molecular-sized or even atomic-sized components [65]. Finally, there is the economical limitation. It becomes increasingly expensive to make faster microprocessors. For example, running microprocessors at high clock frequencies while still guaranteeing that a substantial amount of work is done within one clock cycle, is a hard and especially expensive task.

A very natural solution to reduce the execution time (i.e. the wallclock time) of a program, without being hit by the above limitations, is by performing independent calculations simultaneously, or *in parallel*. So, instead of using a single fast and expensive computer, a bunch of, potentially less expensive, computers could be used to solve a problem. To illustrate this, consider the following example which applies parallelism to people rather than to computers. Imagine yourself being a mailman who has to take care for the distribution of a number of letters. This requires each letter to be stamped after which it must be thrown into the appropriate mailbag (destined for the correct town). However, if there were two people to process the mail, then the letters can be divided among both persons and they can do the work concurrently. Ideally, this halves the time it takes to perform the job for one person. Because of the straightforwardness of this concept and its potential performance gain, many of today's computers exploit parallelism in one or another way.

1.1.1 Parallelism

The concept of parallel computing is not new. In 1842, general Menabrea [90] described the Analytical Engine from Charles Babbage and wrote:

> "When a long series of identical computations is to be performed, such as those required for the formation of numerical tables, the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amount of the processes."

For a long period of time, the technology was not capable of realising these ideas. It wasn't until the late 1960's, with the emergence of (V)LSI technology, that parallel computing became reality. An example of one of these first parallel computers is the Illiac IV [36], which contained an 8×8 array of processing elements. Although this computer was a technological breakthrough (it was the first machine with a large number of processing elements), it failed as a computer due to the fact that it was hard to program and had poor I/O performance. Despite this, the Illiac IV was one of the important achievements that led to the start of the parallel computing era.

Parallelism can be exploited at various levels. That is, the sub-tasks which are computed in parallel can have different *grain sizes*. With respect to this grain size, roughly three types of parallelism can be distinguished:

- *Fine-grained parallelism*
  Parallel computation is exploited at the level of instructions. An example is the *Instruction Level Parallelism* (ILP) exploited by superscalar and VLIW microprocessors [10]. Their compilers and, in the case of superscalar processors, the hardware are capable of scheduling multiple (independent) instructions such that they can be executed simultaneously.

- *Medium-grained parallelism*
  Small or medium-sized sequences of code, such as procedures, are executed on different processing elements. The use of threads, which are light-weight processes sharing a single address space, can be regarded as a form of medium-grained parallelism. An example of a thread-level parallel computer architecture is the Tera MTA machine [4].

- *Coarse-grained parallelism*
  An application is divided in large sub-tasks (potentially containing thousands of instructions) which are executed in parallel. For example, parallel programs adhering to the SPMD (Single-Program, Multiple-Data) programming paradigm apply coarse-grained parallelism [49]. In the SPMD paradigm, every processing element executes a single common program, exploiting implicit parallelism by focusing on its share of the multiple data.

For all three grain sizes, either *data parallelism* or *functional parallelism* can be exploited. In data-parallel programs, such as the ones adhering to the SPMD model, each task performs the same series of calculations but applies them to different data. In functional parallelism, each task performs different calculations using either the same or different data.

1.1.2 Parallelism is not a panacea

The holy grail of parallel computing is to obtain a *speedup* of N when executing an application on N processing elements. We define speedup as the time needed for sequential execution divided by the parallel execution time. In other words, if we double the number of processing elements, then we would like to get twice the performance. When this is true, the parallel application is called to be *scalable*.

It was Gene Amdahl in 1967 who indicated that parallel computing might not have the high performance potential everyone thought it had [5]. In his paper, Amdahl advocated the use of sequential computing rather than parallel computing as the latter form of computation is less cost-effective. This proposition was based on the following observation. When looking at one specific application and a *fixed* program input, linear speedup (i.e. perfect scalability) of the application is often not possible, while the cost of the parallel machine generally does scale linearly (or faster) with the number of processing elements. The incapacity of scaling the application is due to the presence of *sequential components* which cannot be parallelised. To illustrate this, consider Figure 1.2 which shows a possible task graph of a data-parallel program. In this figure, the edges denote the dependencies between the tasks and the numbers within the tasks give the processor number on which a task is scheduled. The length of the critical path between the start and end nodes defines the minimal execution time of the application. Hence, it specifies the upper bound on the speedup. As the start and end tasks of the application in Figure 1.2 are inherently sequential, scaling the

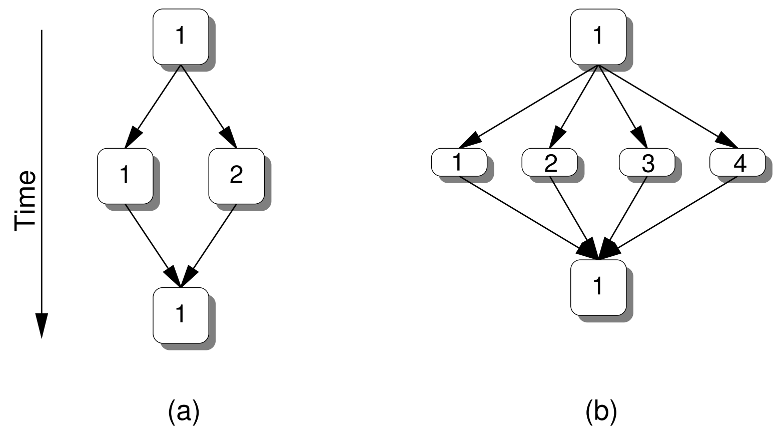


Figure 1.2: A task graph of a data-parallel application.The numbers in the tasks refer to the processors on which the tasks are scheduled. Scaling the application from 2 processing elements (a) to 4 processing elements (b) does not double the performance.

application from 2 to 4 processing elements does not double the application's performance; only the performance of its parallel component is improved, not that of its sequential components. Amdahl formulated this as follows, which is commonly referred to as *Amdahl's law*:

$$Speedup = \frac{1}{s + \frac{p}{N}}$$

where s is the fraction of the program which is inherently sequential, p is the parallel fraction of the program and N denotes the number of processing elements. So, if only 80% of an application can be parallelised, then the upper bound on the speedup is 5, regardless of how many processing elements are used.

The maximal attainable speedup is usually not obtained. The reason for this is that parallel execution typically requires *synchronisation* to coordinate the information exchange between the parallel tasks. This synchronisation process may introduce latencies when parallel tasks are waiting for each other to complete. The overhead of these synchronisations determines at which grain size parallelism can be effectively exploited. Fine-grained parallelism, for instance, requires frequent synchronisation, basically after executing one or only a few instructions. Therefore, the synchronisations must be cheap. On the other hand, synchronisations in coarse-grained parallel applications are much less frequent. As a result, these applications can tolerate higher synchronisation overheads.

Consider again the example in which a bunch of letters has to be stamped and ordered according to the town of destination. Figure 1.3 shows the task graph of a possible parallel algorithm for this application. Note that this algorithm is not meant to be the most efficient one but it is only used for the purpose of illustration. Initially, processing element 1 distributes the letters evenly over processing elements 1 to N. Each of these processing elements stamps the letters and deposits them into a new mailbag, called the *stamp-bag*. There is one private stamp-bag per processing element. When the stamp-bag of a processing element is full or there are no letters left to be stamped, the stamp-bag is sent to one of the elements $N+1$ to M and the the process of stamping can be continued using a new, empty stamp-bag. Subsequently, the processing elements $N+1$ to M sort the letters from the incoming stamp-bags. To do so, they deposit each letter into a container which is headed for

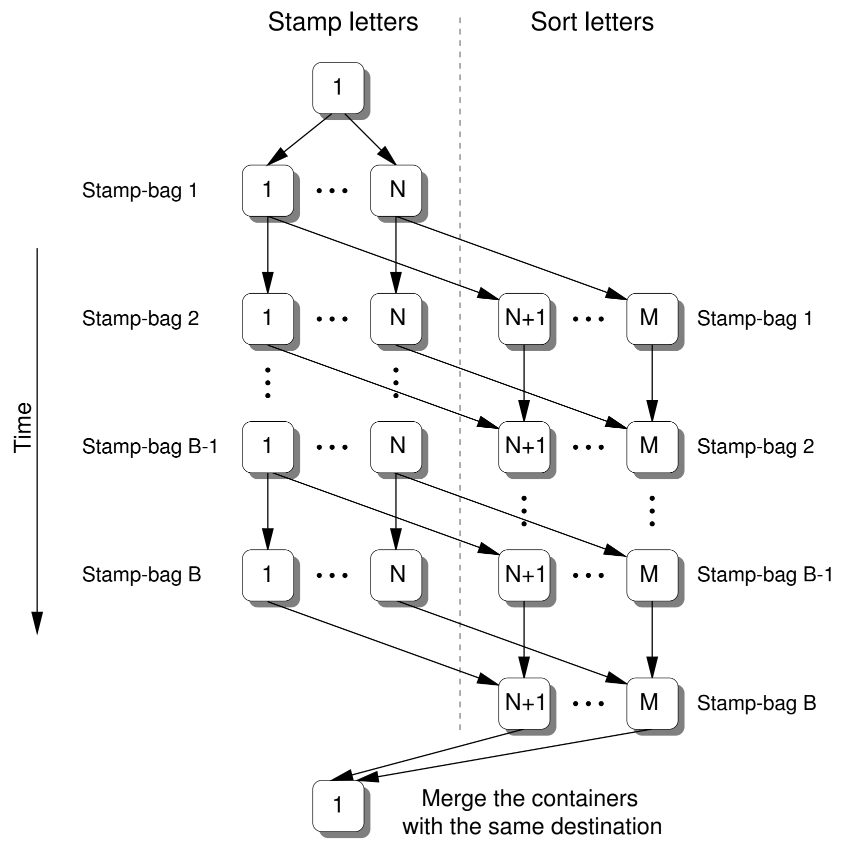


Figure 1.3: Stamping and ordering mail: a possible parallel algorithm.

the appropriate town. There is one container per destination for each of the elements $N+1$ to M. Finally, when all sorting is done, processing element 1 receives the per-town containers from elements $N+1$ to M and merges the containers that are destined for the same town. If there are B stamp-bags stamped and sorted, then the critical path of this algorithm consists of $B+2$ steps of which there are B parallelised.

The process of synchronisation is clearly shown in Figure 1.3: a stamp-bag cannot be sorted until it is received from the processing elements which stamp the letters. For this reason, elements $N+1$ to M are running one step behind with respect to the elements 1 to N.

Scaling this algorithm to a large number of processing elements could result in high synchronisation overheads. Imagine, for instance, the extreme case where there are 100 people to process 100 letters (i.e. N = 50 and M = 100). So, each person receives one letter after which he or she can forward the letter immediately. In such a situation, a substantial amount of time is spent on exchanging the letters rather than on stamping and ordering them. A potential solution to this problem is by scaling the data size on which the computations are performed, e.g. by increasing the number of letters which have to be processed. This brings us to another famous definition of scalability, namely that of Gustafson [50]. This definition essentially states that doubling the number of processing elements should allow the computation of *twice the problem size* within the same time. So, rather than keeping the problem size constant like Amdahl's definition dictates, Gustafson's definition of scalability assumes that the problem size is scaled as well. This implies, however, that Gustafson's definition can only be applied to data-parallel applications as they allow the problem size to be easily scaled.

Evidently, computer architectures may benefit from parallelism in order to yield higher performance, but parallel computing is certainly not a panacea. Although the basic concept of parallelism is straightforward, its effective exploitation is far from trivial. For this reason, a lot of research has focused on finding techniques to automate the search for and exploitation of parallelism. In some areas, this research effort has led to commercially viable products. For example, most of today's microprocessors are superscalar. These processors dynamically issue instructions to their multiple execution units, thereby exploiting Instruction Level Parallelism (ILP) at run-time. Basically, this process is hidden from the programmer, who is therefore not bothered by the tedious task of finding the parallelism. Another, and slightly older, success story is *vector processing*. For computer systems containing vector co-processors [52, 143] (which will be described in the next section) and vectorisable applications, current compiler technology is sufficiently mature to generate code that effectively utilises the vector processors. As a consequence, vector computing has been an accepted and widely-used parallel computing technique for several decades now. However, because of their high cost, vector computers are more and more superseded by less expensive MIMD parallel platforms, which will be discussed in the next section.

In the areas where more coarse-grained parallelism is involved, the exploitation of parallelism is less implicit. In these cases, the programmer may have to invest a substantial amount of time to restructure or reprogram applications in order to make them suitable for parallel execution. Several efforts have been made, and are still being made, to assist the programmer in this process of parallelisation. For instance, to improve the portability of parallel applications, products such as PVM [136] and the MPI [41] standard provide a generic interface for the creation of and communication between (coarse-grained) parallel tasks. As both PVM and MPI implementations are widely available, the applications using their interfaces can more or less be readily executed on a wide range of parallel platforms.

Another example is High Performance Fortran (HPF) [54], which is a data-parallel SPMD language. In this programming language, the programmer explicitly specifies both the distribution of the data over the processing elements and the parallel operations that must be applied to the data. The compiler automatically translates the memory references to non-local data (residing in some remote memory) into communication primitives which request the remote data.

Despite of all these initiatives, many of the techniques and tools are still not sufficiently mature to provide the desired basis for the development of coarse-grained parallel applications. Programming these applications still requires the programmer to be highly skilled, especially in comparison with traditional sequential programming. So, there remains room for improvement in order to make the exploitation of coarse-grained parallelism more user-friendly.

1.2 Parallel computer architectures

Having described several different forms of parallelism, a similar distinction can be made for the computer architectures which actually perform the parallel computations. An old but still widely-used classification of computer architectures was proposed by Flynn in 1972

		Data streams	
		Single	Multiple
Instruction streams	Single	SISD	SIMD
	Multiple	MISD	MIMD

Table 1.1: Flynn's taxonomy of computer architectures.

[39]. This classification takes two dimensions into account: the *instruction stream* and the *data stream*. The first dimension indicates how many instruction streams are executed in parallel, while the second dimension specifies how many data streams are operated upon in parallel. As a result, four types of architectures can be distinguished, which are listed in Table 1.1.

Pure sequential computers (i.e. traditional Von Neumann architectures) belong to the class of SISD (Single-Instruction, Single-Data) computers. They execute one instruction stream which processes a single data stream. Architectures in the remaining three classes contain multiple processing elements. In the class of MISD (Multiple-Instruction, Single-Data) computers, multiple processors operate on a single data stream. Only a few actual examples of these computer architectures exist, like the special-purpose systolic arrays [73]. The MISD class was included more for the sake of orthogonality than to identify a group of actual computers.

Computers from the SIMD (Single-Instruction, Multiple-Data) class have been around for about 30 years now and, to a certain extent, quite successfully. As its name already suggests, a SIMD architecture is inherently data-parallel. Basically, this class can be subdivided into two categories: *vector computers* and *array processors*. A vector computer contains pipelined vector units which allow operations on whole vectors (arrays of possibly hundreds of data elements) at once. Vector computers are considered to be SIMD because the various stages of a vector pipeline are simultaneously working on different data elements within the vector. Well-known examples of vector machines, which are generally used for scientific number-crunching applications, are the traditional supercomputers. The most famous of these machines include, of course, the ones that belong to the Cray family, such as the Cray Y-MP.

The alternative class of SIMD architectures, namely that of array processors, is somewhat different. These machines typically consist of a large number of, often simple, processors which are connected in a network to exchange data with each other. All of these processors synchronously execute the same instruction in parallel, where each processor applies the operation to its own (local) data elements. The application domain of this class of architectures is usually restricted to scientific computing and image processing. The Thinking Machines CM-2 is an example of an array processor which exploits fine-grained parallelism [143]. It contains up to 2^{16} processing elements that can perform bit-serial operations. Naturally, more coarse-grained array architectures also exist, such as the TM CM-5 which contains up to 2^{14} 32-bit (and 64-bit floating point) Sparc processors [125, 143].

Nowadays, the MIMD (Multiple-Instruction, Multiple-Data) architecture is probably the most popular type of parallel machine. Unlike SIMD architectures, MIMD computers allow asynchronous operation in which each processor executes its own program (instruc-

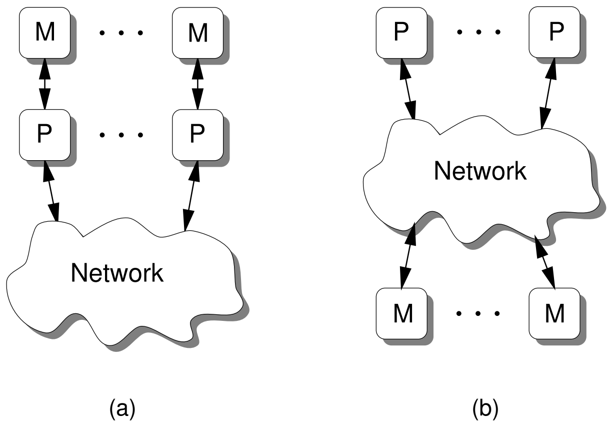


Figure 1.4: The two memory organisations for MIMD architectures: a distributed memory multicomputer (a) and a shared memory multiprocessor (b).

tion stream). Consequently, both functional and data parallelism are supported. Generally, a MIMD machine is referred to as a medium to coarse-grained parallel platform in which a number of processors are connected to each other in a network. However, fine-grained parallel superscalar and VLIW processors can also be regarded as (synchronous) MIMD architectures. Nevertheless, throughout this thesis, we only use the term MIMD to refer to the more coarse-grained architectures.

The class of MIMD architectures can be divided into two categories based on the type of memory organisation: *distributed memory* machines and *shared memory* machines. This is illustrated in Figure 1.4. In distributed memory computers, which are commonly called *multicomputers*, each processor has its own local memory (see Figure 1.4a). If a processor needs to access data which resides in another processor's memory, then the data must be explicitly transferred by sending a message from the remote processor to the requesting processor. For this reason, these architectures are often called *message-passing* machines. The big advantage of this type of architecture is that it is generally scalable to hundreds or even thousands of processors. In the case of such a large machine, the term Massively Parallel Processor (MPP) is commonly applied. But, on the other hand, the communication between processors in multicomputer machines is quite expensive and often not transparent. With the latter, we mean that the message-passing communication usually has to be programmed explicitly. Naturally, a few exceptions exist, such as when using the previously mentioned HPF programming language. An example of a popular multicomputer is the IBM SP-2, which consists of RS/6000 processors connected in a multi-stage network [125].

In shared memory MIMD architectures, or *multiprocessors*, the programmer accesses the memory using a single linear address space (see Figure 1.4b). The shared memory may reside at one central place (implemented as one memory or as a set of memory banks) or it may be physically distributed such that a *node* of one or more processors "owns" a part of the shared memory. In a configuration where the memory is centralised, the access characteristics of the memory are identical for each processor. Therefore, these architectures are called *Uniform Memory Access* (UMA) machines. If all the system's resources can be accessed in a uniform manner, then the term *Symmetric MultiProcessor* (SMP) is often used.

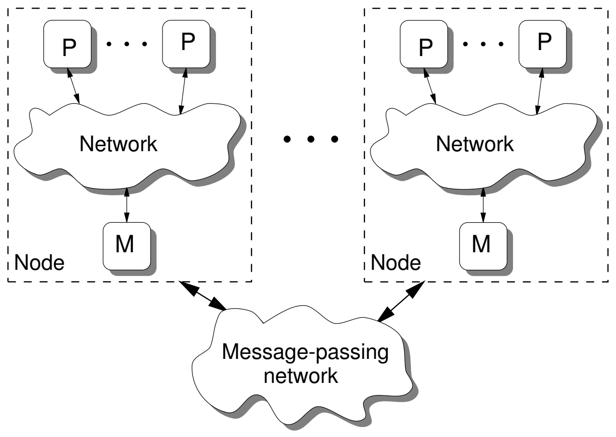


Figure 1.5: The structure of a NUMA multiprocessor architecture.

For a long time, most multiprocessors were UMA machines. However, the problem with these platforms is their poor scalability. It is often too expensive or even impossible to scale the shared network of UMA platforms to support a large number of processors. If, for example, a bus is used for the interconnect between a large number of processors and the shared memory, then the contention on the bus could become so high that it makes the machine come to a halt. Of course, caching will help to reduce the bus traffic, but it does not allow to increase the number of processors endlessly.

To be more scalable, current shared memory MPPs typically distribute the memory over their nodes. The nodes, which may contain multiple processors, are subsequently connected to each other by a message-passing network. This is shown in Figure 1.5. These platforms are called *virtual shared memory* machines as the memory is physically distributed but logically shared. The difference between a virtual shared memory machine and a traditional distributed memory multicomputer is that the programmer of the first machine still accesses one linear address space. Memory references to non-local data are handled either by the hardware or the Operating System (OS), which fetches a copy of the required data by means of message passing. This is all done transparently to the user. However, as references to local data only need a simple memory lookup and whereas the references to non-local data require slower message-passing communication, there is an imbalance between the latencies of local and non-local memory accesses. Therefore, these architectures belong to the so-called class of *Non-Uniform Memory Access* (NUMA) machines. Because of the imbalance between the latencies of local and remote memory references in NUMA architectures, extra care has to be taken when programming these machines. More specifically, the programmer should try to limit the number of accesses to remote data. So, the aspect of *locality* is even more important for NUMA machines than it is for UMA machines.

Like in UMA machines, the use of caches can reduce the network traffic as, for example, copies of remote data may be alive in the local data cache for quite a while. However, using caches in both UMA and NUMA architectures requires the consistency of the caches to be guaranteed. If some data is read from a particular cache, then this data is supposed to be up-to-date. At the time a data element is retrieved from the local cache, there may not exist a newer (i.e. a more recently updated) version of that data element in another cache. To accomplish this, a *cache coherency protocol* is required which guarantees the consistency of the, possibly replicated, cached data [133]. For instance, NUMA machines that apply these protocols are called cache-coherent NUMA, or ccNUMA, architectures.

Some NUMA architectures extend the amount of caching to its logical extreme. In these so-called COMA (Cache Only Memory Architecture) machines, such as the KSR1 [72] and the Data Diffusion Machine [141], the entire linear address space is spanned by associative memories (i.e. caches) only. As a consequence, data elements do not have a fixed home location and dynamically migrate when they are requested.

An example of a cache-coherent UMA machine is SGI's Power Challenge. This machine is a bus-based multiprocessor that is capable of holding up to 18 MIPS R8000 processors [125]. The more recent multiprocessor from SGI, the Origin 2000, is a ccNUMA machine [80]. It scales up to 512 nodes containing one or two MIPS R10000 processors and 1 TB of memory in total. The nodes of this machine are connected in a hypercube network by means of 800 Mbyte/s Cray-links.

As the performance of modern microprocessors reaches several hundreds of MFLOPS, today's MPP multicomputer and multiprocessor systems are capable of obtaining a peak performance of hundreds of GFLOPS, or even 1 TFLOPS (the ASCI Red UltraComputer powered by thousands of Intel Pentium Pros [142]). Although this peak performance is often not obtained in reality, parallel platforms typically provide a large amount of computational power. Clearly, such computational power can, for instance, be exploited to help solving the so-called "Grand Challenge" problems. These well-known science and engineering problems are extremely demanding with respect to their computational needs and often to their memory consumption as well. Hence, it is unthinkable that they can be solved using current uni-processor technology.

1.3 Organisation of this thesis

In this dissertation, we address the performance evaluation of computer architectures. More specifically, we focus on the simulation techniques used for evaluating the performance of parallel computers and, in particular, multicomputer architectures.

The remainder of this thesis is divided in two parts. The first part consists of three chapters which deal with the simulation techniques and tools used for the performance evaluation of parallel computers. The first of these chapters, Chapter 2, gives a general overview of the field of computer architecture simulation. It describes several popular simulation methods, such as trace-driven and execution-driven simulation, and discusses the issues playing a role in the different simulation approaches.

Chapter 3 presents a novel simulation environment, called *Mermaid* [108, 109, 114], which supports the performance evaluation of multicomputer and, to some extent, multiprocessor architectures. Mermaid differs from other simulators in the way it addresses the tradeoff between simulation performance, flexibility and accuracy. For example, by lifting the simulator's detail to a level which is higher than the regularly-used instruction level, a flexible architecture evaluation framework is obtained. The work discussed in this chapter has been greatly influenced by the work of Henk Muller [97] who has, among other things, developed the simulation language we use to implement our architectural simulators.

In Chapter 4, we show that Mermaid's flexibility has hardly compromised the simulation performance and accuracy. To do so, we have studied a simulation model of an available multicomputer architecture using a suite of benchmark programs. In order to validate the model, the simulation results are compared with the execution results from the real machine. Furthermore, to get a feeling of where Mermaid stands performance-wise, we compare its efficiency with that of several other, state-of-the-art parallel architecture simulators.

Chapter 4 also describes an extension to our simulation environment which makes it possible to perform distributed simulation on a cluster of workstations. We have evaluated the performance of this enhanced simulator and found that the distributed simulation may significantly boost both the performance and the scalability of the simulation.

The second part of this thesis contains three case studies in which the Mermaid simulation environment was used. Chapter 5 presents a performance evaluation of a wormhole-routed Mesh of Clos network. This network forms the interconnect of a series of Parsytec multicomputers that was introduced a few years ago. With the evaluation study, which was performed before the actual machines were built, insight was gained into the potential communication performance of this particular multicomputer architecture.

In Chapter 6, we discuss the application of Mermaid in the Philips PROMMPT project. This project aims at the development of a successor of the Philips TriMedia TM-1 VLIW processor [127]. We explored a part of the design space of this new processor. More specifically, we studied several prefetch techniques for TriMedia's data cache in order to reduce the average memory latency.

Chapter 7 describes the modelling work that has been performed in the scope of the IMPACT project [57], which was launched to stimulate the research on parallel computing methods for large-scale databases. Our participation in this project consists of investigating the performance behaviour of a particular scalable, distributed data structure for a number of multicomputer network architectures. This distributed data structure will eventually form the heart of a parallel version of the Monet database system [15], which is to be developed at the CWI (this is the Dutch national institute for mathematics and computer science).

In the concluding chapter, Chapter 8, we look back on and discuss the work which was carried out. Additionally, we mention some possible future research.

Part I

Simulation of parallel computer architectures

Chapter 2

Computer architecture simulation

"Door meten tot weten (To knowledge by measurement)."
Heike Kamerlingh Onnes

The concept of modelling and simulation is applied in many disciplines of engineering and science. It is used in the analysis of physical systems in order to gain a better understanding of the functioning of our physical world. Or, to quote Bernard Zeigler [144]:

> "Modelling means the process of organising knowledge about a given system."

Basically, modelling and simulation can be used from two different kinds of perspective, as was mentioned by François Cellier in [20]:

> "It can thus be said that modelling is the single most central activity that unites *all* scientific and engineering endeavors. While the scientist is happy to simply *observe* and *understand* the world, i.e., create a model of the world, the engineer wants to *modify* it to his advantage. While science is all *analysis*, the essence of engineering is *design*."

So, simulation can be used for both analysis (the scientist's point of view) and for design (the engineer's point of view). In this thesis, we take an engineering point of view as we address the *design* of computer architectures.

Modelling and simulation can be characterised as the complex of activities associated with constructing models of real world systems and simulating them on a computer. Generally, this involves two basic steps [115]:

- *Model development*
  The construction of a model that represents all of the important aspects of the system. This also includes the validation of the model in which it is determined whether or not the model is a representative reflection of the reality.

- *Experimentation*
  The design and evaluation of experiments using the model. Simulation allows experimentation with systems that do not exist, or for which it is infeasible to perform the actual experiment.

In the first step, the model development, a lot of intuition is involved. The modeller should, for instance, decide at which *abstraction level* a given system is modelled. Some system components may not be of great interest and can therefore be modelled at a high level of abstraction, which corresponds to a low level of detail. Similarly, important system components can be modelled at a lower level of abstraction (i.e. at a higher level of detail) in order to improve the simulation's accuracy. Unfortunately, increasing the level of detail of a model causes the simulation to be more computationally intensive. Hence, there is a trade-off between simulation accuracy and performance.

Placing the above in the context of performance evaluation of computer architectures, the architect should decide at which abstraction level the performance of the computer architecture components is modelled. Should the model, for instance, be capable of explicitly simulating each separate machine instruction from an application or should it try to estimate the performance of a whole group of instructions (e.g. a basic block [3]) at once? Simulation of separate machine instructions is, of course, more accurate but also more computationally intensive than simulating whole groups of instructions at once.

Furthermore, it should also be decided which components of the architecture may affect the performance and should therefore be included in the model, and which components can be left out of the model. The significant involvement of the architect's intuition and expertise in all of these decisions indicates that the performance modelling of computer architectures is, like computer architecture design itself, more an art than a science.

After a model has been developed, it must be *validated*. In other words, it must be checked if the model is a representative reflection of the real system for the area of interest. We define the area of interest as the set of experiments performed using the model. Generally, no model of a system is valid for *all* possible experiments except the real system itself or an identical copy thereof. So, like François Cellier nicely stated in [20], the modeller should take care *not to fall in love with the model*:

> "All too often, simulation is a love story with an unhappy ending. We create a model of a system, and then fall in love with it. Since love is usually blind, we immediately forget all about the experimental frame, we forget that this is *not* the real world, but that it represents the world only under a very limited set of experimental conditions (we become 'model addicts')."

Strongly related to the model construction is the "model execution", being the actual simulation. Roughly, two types of simulation can be distinguished: *time-driven* and *event-driven* simulation. In time-driven simulation, the simulated system is studied as a function of the time. This simulation class can be further divided into *continuous time* and *discrete time* simulation. Continuous time simulation uses differential equations to describe the system's behaviour over time [128]. This type of simulation is often applied in low-level simulation of analog hardware (i.e. *micro-architecture simulation*), like in transistor-level models [20]. In discrete time simulation, the system under investigation is studied using a finite number of discrete time steps. An example of this approach is a microprocessor model in which every separate cycle is simulated. Here, a discrete time step is equal to the processor cycle time.

Although time-driven simulation is well suited for the functional evaluation of computer architectures, it is often less suited for studying the performance. In many occasions, the performance of computer architectures can be accurately modelled at an abstraction level

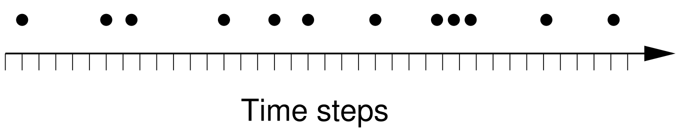


Figure 2.1: A typical example of the “interesting” events occurring in a computer system as a function of the time. The dots refer to the events.

which is much higher than, for instance, the transistor-level. This is often referred to as *macro-architecture simulation*. Due to the increase of abstraction level, the events which are “interesting” for the performance evaluation of a computer architecture, and which should therefore be modelled, do typically not occur at every possible time step. Instead, they take place at arbitrary, non-uniform time steps. This is illustrated in Figure 2.1, in which the dots denote the interesting events. When applying time-driven simulation under such circumstances, the time steps at which no interesting event occurs are also simulated, inducing a lot of redundant overhead. Consider, for example, the time-driven simulation of the microprocessor again. In many simulated cycles, such as the ones in which the processor is stalled, no interesting activity takes place. As a consequence, these cycles do not need to be simulated explicitly.

To avoid these redundant simulation overheads, performance models often apply event-driven simulation, which is commonly referred to as *discrete-event simulation*. In this simulation technique, only the events that actually contribute to the performance estimation of the architecture are simulated. As a result, the execution time is linear to the number of simulated events, which often is more efficient than time-driven simulation (which has an execution time that is linear to the number of simulated time steps). The decision of which events are interesting to model and which are not, largely depends on the abstraction level of the model. If, for example, the performance of a data cache is modelled, then only memory reference events defining the type of operation (read/write) and the data address need to be simulated.

All of the simulation techniques discussed in this thesis are based on discrete-event simulation. The differences between them are mainly related to the abstraction level at which the simulation operates (i.e. the type of events which are used) and the approach of generating and consuming the events. In the remainder of this chapter, we present an overview of the most popular techniques to simulate computer architectures.

2.1 Trace-driven simulation

The implementation of a computer architecture simulator can be divided into two components: an event generator and a post-processor. The event generator produces a *trace* of execution events which are consumed by the post-processor. Subsequently, the post-processor handles the incoming trace of events with the required degree of interest: it may simply keep behavioural statistics or it may perform a simulation of the *target* architecture in order to estimate, for instance, the timing consequences of the events. With the term “target architecture”, we refer to the architecture that is being evaluated. If the post-processor is a simulator, then we call this technique *trace-driven simulation* [138]. This simulation approach is typically used for studying the performance of memory hierarchies in uni-processor systems.

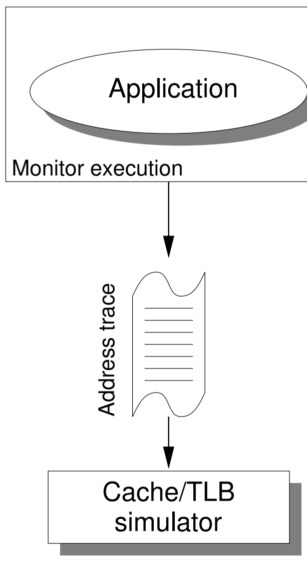


Figure 2.2: Trace-driven simulation of memory behaviour.

Consider, for example, Figure 2.2 in which trace-driven simulation of memory behaviour is illustrated. The data address references of an application, and possibly the instruction references, are captured by monitoring the execution of the application. Subsequently, the obtained address trace is forwarded to the simulator which can perform a cache or TLB (Translation Lookaside Buffer) simulation to gather statistics on, for example, the missrate of the cache or TLB. In this scheme, the application can basically be executed and monitored on an arbitrary computer to gather the address trace. However, as we will explain later in this chapter, doing so for the study of parallel machines requires extra measures to be taken.

The major advantage of trace-driven simulation is the separation of the trace generation and trace consumption. This simplifies the implementation of the simulator and allows the results of a trace generation run to be stored (e.g. on disk) in order to be re-used with different simulators (i.e. post-processors). Storing the generated traces may, however, consume large amounts of storage. For this reason, much research effort has been put in finding techniques to reduce the size of traces. A brief overview of these techniques is given later in this section. First, we will discuss the methods which are commonly used for collecting the traces. As trace-driven simulation is mostly used for evaluating memory behaviour, our focus in the remainder of this section is on *address tracing*.

2.1.1 Trace collection

For accurate simulations, it is essential that the extracted address trace closely resembles the actual memory behaviour of an application. Trace quality can be measured by its *completeness*, level of *detail* and *distortion*. A *complete* trace includes the memory references made by all components of the system, including those made by, for instance, the OS. In a *detailed* trace, the trace events contain enough information to allow accurate reproduction of the execution. This is, for example, not possible when a trace only contains raw addresses and lacks information on the type of operations (read/write). Finally, a trace should be *undistorted*. This means that it should not include any memory references which do not appear

Tracing technique	Completeness	Detail	Distortion
Hardware probes	good	medium	no—low
Microcode	good	high	medium
IL-simulation	poor	medium—high	no
Code annotation	poor	medium	high

Table 2.1: Characteristics of the different trace-collecting methods.

due to the execution of the application but are caused by the tracing process. Moreover, the addresses in the trace should appear in the same order as the actual references. Two well-understood forms of distortion are *time dilation* and *space dilation* which occur when the tracing method causes the monitored application to slow down (time dilation) or to consume more memory (thrashing the cache) than it normally would (space dilation).

The collection of an address trace can be performed at various levels using different methods. The most well-known of these methods are:

- External hardware probing
- Microcode modification
- Instruction-level simulation
- Code annotation

As is shown in Table 2.1, each of these trace-collecting techniques has different characteristics with respect to the completeness, level of detail and distortion of the resulting trace.

In hardware probing [38], probes are connected to the processor pins in order to record their activity. The address and control signals are fed into an external memory buffer, and when this buffer is full, its contents are transferred to a storage device, such as disk. Clearly, this method generates a complete trace. If the generated trace can be stored entirely, then the trace is free from distortion. This is, however, not trivial as the occasional emptying of the buffer basically requires that the rest of the system is stalled. A good discussion on this issue can be found in [138]. Moreover, the level of detail is not exceptionally high as the captured hardware events may be hard to interpret. It is, for example, hard to relate a memory reference to the process that made it. The main disadvantage of hardware probing is, however, its high cost since logic analyzers are quite expensive.

In microcode-based trace collection, which was introduced by ATUM [2], the microcode of a processor is modified to allow the tracing of memory references. Like hardware probing, this method is complete as all memory references (including those from the OS kernel) go through the microcode. Moreover, at the microcode level, enough information is available to produce a detailed trace. This method does suffer, however, from small distortions. Because instructions take longer to execute (due to the extra microcode), external devices (e.g. disks) appear to the application to be faster than they actually are, and interrupts from the system clock occur more frequently. As a consequence, the traced application behaves differently compared to its original behaviour. The primary drawback of microcode-based tracing is that it is obsolete. Many of today's microprocessors are based on the RISC principle and therefore have hardwired control rather than microcode.

Similar to microcode instrumentation, instruction-level simulators can also be modified to collect address traces. An instruction-level simulator interprets executable images, which are written in the target instruction set, and executes them by *emulating* the hardware [25, 77]. In other words, it executes one Instruction Set Architecture (the target ISA) in terms of another ISA (that from the host on which the simulation is running). A more elaborate description of these simulators is presented later in this chapter.

Usually, instruction-level simulators do not trace the OS kernel references and are therefore not capable of generating complete traces. Most of the simulators that claim to trace kernel references only intercept and emulate system calls. They do not trace the actual references of the OS. There are, however, a few exceptions, such as SimOS [119, 118]. This simulator allows for booting a fullfledged OS on top of it and is therefore capable of tracing the actual OS references. To do so, the complete computer system must be simulated, including the devices such as disks.

With respect to the trace detail, instruction-level simulators typically operate with virtual addresses and they are unable to determine the actual physical addresses (naturally, this is not true for SimOS). In contrast to microcode-based tracing, simulator-based tracing is non-intrusive which implies that the resulting trace does not suffer from distortion. This is because a simulator uses its own (virtual) simulation clock for timing purposes, which is not affected by the trace collection code.

The major disadvantage of this type of trace collection is that instruction-level simulation generally is inefficient. On the other hand, this method is more flexible and portable than the two previously discussed methods.

Finally, if the target and host instruction sets are identical, then an application can be statically annotated to record its behaviour rather than dynamically interpreting it by an instruction-level simulator. With this technique, instructions are inserted around memory references to create a new executable that produces an address trace whenever the application is executed [35, 79, 76, 132].

Most code annotators are incapable of tracing kernel references and therefore do not generate complete traces. However, in some systems, such as *Epoxie* from Borg et al. [17], parts of the OS are instrumented as well, supporting the collection of kernel references. Nevertheless, static code annotation does not allow the tracing of, for example, dynamically-linked or dynamically-compiled code. Moreover, like the instruction-level simulators, code annotators often have difficulties with respect to the calculation of physical addresses. But, in contrast to simulation-based trace collection, the code annotation method is intrusive and therefore suffers from distorted traces. Evidently, these distortions are caused by the inserted instructions.

Compared to the instruction-level simulators, code annotators are often more easy to implement, faster and about equally flexible and portable. For these reasons, code annotation is, despite its poor trace quality, the most popular form of trace collection. This clearly illustrates the awkward tradeoff in trace generation: the trace collecting techniques that produce high quality traces are either obsolete or too expensive, while the flexible and low-cost techniques only deliver modest trace quality.

2.1.2 Trace reduction

Modern superscalar microprocessors, running at high clock frequencies, may generate address traces of hundreds of megabytes per second. Clearly, storing and processing such large traces is a problem. The simplest way to solve the storage problem is by not storing the traces at all. Some trace-driven simulators generate and process the traces *on-the-fly*. This technique is most effective when a trace can be generated at approximately the same speed as it can be read from disk. A prerequisite of this approach is that the traced application is available. There may, however, be situations in which only the trace and not the application can be made available. We encountered a similar situation in the PROMMPT project (see Chapter 6) in which the traces were generated at Philips and simulated at our department. In that case, one may have to fall back on explicitly storing the traces again.

Due to the above problems, there is a need for finding ways to actually reduce the enormous size of traces to minimise both processing and storage requirements. Generally, two classes of trace reduction approaches can be distinguished. The first class consists of techniques that trade efficiency for smaller traces. These techniques require extra processing time in order to reproduce the original trace from the reduced trace before the simulation can be performed. In the second class of reduction methods, trace quality is traded for smaller traces. Hence, these techniques produce smaller but *incomplete* and *distorted* traces. Simulating these incomplete and distorted traces results in a simulation error. Provided that the error is kept reasonably small, this type of trace reduction might be applicable in some evaluation studies. Table 2.2 gives an overview of several widely-used reduction methods and presents their individual characteristics. The numbers listed in this table are *typical values* and were taken from [138]. The reproduction slowdown indicates the slowdown factor of reproducing the reduced trace into the full trace compared to just reading the full trace from disk. Furthermore, the simulation speedup refers to the number of times the trace processing (i.e. simulation) performance has improved due to the smaller size of the filtered/sampled traces.

The most straightforward approach to reduce the trace size is by applying a standard data-compression algorithm, such as Lempel-Ziv compression [145]. Alternatively, a *relative* trace can be generated [121, 61]. In such a relative trace, offsets are used rather than full addresses. To illustrate this, consider Figure 2.3 in which a trace is converted to a relative trace. The first instruction reference contains the full address, after which each succeeding instruction reference is relative to its predecessor. Due to the spatial locality of the data references, an identical scheme can also be used for the load and store operations. Alternatively, *page addresses* can be used comprising, for instance, the 16 higher-order data-

Reduction technique	Reduction factor	Reproduction slowdown	Simulation speedup	Error
Trace compression	10-100	100-200	—	—
Significant-event traces	10-40	20-60	—	—
Trace filtering	5-100	—	4-50	<15%
Trace sampling	5-20	—	<10	<10%

Table 2.2: Trace reduction techniques.

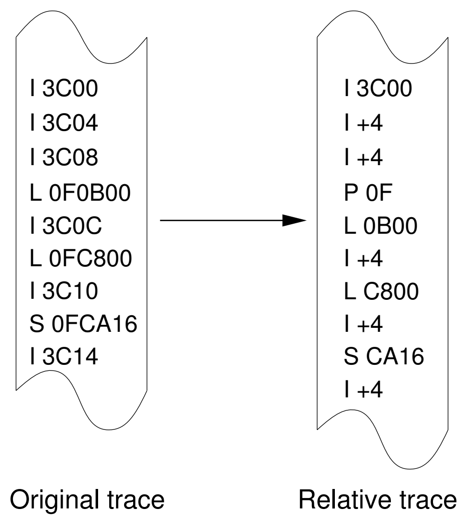


Figure 2.3: Trace compression by calculating a relative trace. An *I* refers to an instruction fetch, an *L* to a load, an *S* to a store and a *P* to setting a page address.

address bits. In this scheme, which is illustrated in Figure 2.3, the load and store events only need to specify the 16 lower-order bits of a data address. The simulator subsequently calculates the full data addresses by OR'ing the addresses of the load/store events with the active page address.

Applying relative traces has two advantages. First, relative traces are, of course, smaller than the original ones. Second, when used in combination with a Lempel-Ziv algorithm, even higher compression rates are obtained. This is because of the large amount of regularity within the relative traces due to, for instance, the striding patterns. On the other hand, the main drawback of trace compression techniques is that the original trace has to be reproduced just before simulation, inducing a slowdown which may be substantial (see Table 2.2). Generally, all trace compression techniques are *lossless*. This means that the compressed trace has the same information quality as its original trace. So, there is no simulation error.

Another way to achieve lossless trace reduction is by generating a significant-event trace. An example of such a trace system is AE [76]. In AE, only a small set of events, the so-called significant events, is recorded. These events serve as input to an abstract version of the program that reproduces a full trace by mimicking the portions of the original program that involve address calculation and memory referencing. In this scheme, the significant events are the addresses which are hard or impossible to determine statically by the abstract program (like pointer parameters in functions).

Trace filtering is an approach that reduces both the size of traces and the time that is needed to process them. This technique is, however, not lossless and therefore results in a simulation error. Two well-known trace filtering methods are *stack deletion* and *taking snapshots* [130]. In stack deletion, a stack is maintained containing the last D memory references of the full trace. Subsequently, the memory references that hit in the stack are discarded from the original trace while the references that miss are concatenated to form the reduced trace. Similarly, when taking snapshots, the reduced trace only consists of every Nth reference from the original trace. Naturally, more complex filtering techniques exist of

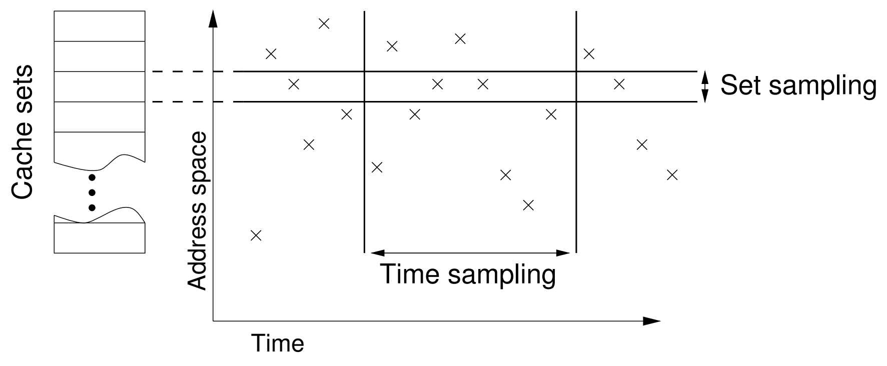


Figure 2.4: Time and set sampling of a trace.

which an overview can be found in [138].

Instead of occasionally throwing away references from the original trace, a subset *sample* of the original trace can also be taken. Figure 2.4 shows two types of trace sampling: *time sampling* and *set sampling*. In this figure, the crosses represent the memory references which are visualised according to their spatial distribution (vertical axis) and their temporal distribution (horizontal axis). Time sampling is performed by selecting segments of references that occur during some time interval [74]. This method should, however, be performed with care. A sufficient number of trace segments must be collected to ensure that the memory behaviour is adequately represented. Moreover, this trace reduction method amplifies the so-called *cold-start bias* of simulated caches: at the beginning of every sampled trace segment, the cache does not know whether the initial references are hits or misses.

In set sampling [116], only the memory references are selected that map to one or more specified sets of a set-associative cache [131]. Subsequently, simulating only the cache sets which are present in the trace gives an estimate of the overall cache performance. Unlike time sampling, set sampling does not suffer from a large cold-start bias as the sampled sets include all references of the original trace. However, a difficulty in set sampling is that the produced trace is more or less dependent on the cache configuration (i.e. its associativity). There are methods to overcome this problem [138], but their description is beyond the scope of this thesis.

2.1.3 Optimising trace-driven simulation

Accurate trace-driven simulation of memory behaviour may be surprisingly slow. In a paper by Gee et al. [44], simulating the cache behaviour of the SPEC92 benchmark suite [99] is reported to take *7 months* of calendar time when running the simulations on seven workstations. This study also showed that a large amount of processing time is wasted in the common case. Addresses which hit the cache (the common case) are traced in the same fashion as the addresses missing the cache. The cache hits, however, do not require any action from the cache simulator in the case of a direct-mapped cache or a set-associative cache with random replacement. Moreover, hits to the MRU (Most Recently Used) cache block within a set of a set-associative cache with LRU replacement do, again, not require any action. For this reason, the common case can often be optimised. Two of such optimisations are *active memory* [82] and *trap-driven simulation* [137]. Both are on-the-fly simulation techniques,

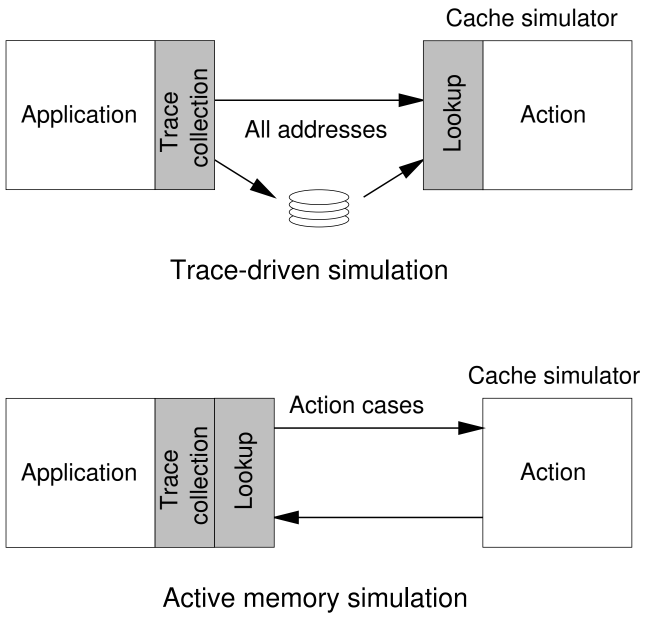


Figure 2.5: Trace-driven simulation (top) versus active memory simulation (bottom) of a cache.

thus generating and processing the address trace at the same time. Hence, they rely on the fact that the trace collection methods have improved to the point that regenerating the trace is nearly as efficient as reading it from secondary storage [78].

In active memory, the memory is logically partitioned in user-defined blocks to which a state can be attached. The state of a memory block determines whether or not it is resident in the simulated cache. A large table, which stores the memory block states, is used by the trace collection mechanism to swiftly determine if there is a cache hit or not. So, the cache lookup is performed by the trace collector rather than by the cache simulator. According to the state of an accessed memory block and the type of reference, a user-specified handler routine is invoked. During the simulation, the state of the memory blocks is manipulated to control which of the handlers is invoked when referencing a particular block. If the state of a memory block is valid at its reference, indicating that it is present in the cache, then a so-called *NULL* handler is invoked. This NULL handler specifies the common, no-action case which can be processed quickly as it does not invoke the complete cache simulator. In Figure 2.5, the difference between traditional trace-driven simulation and active memory simulation is illustrated.

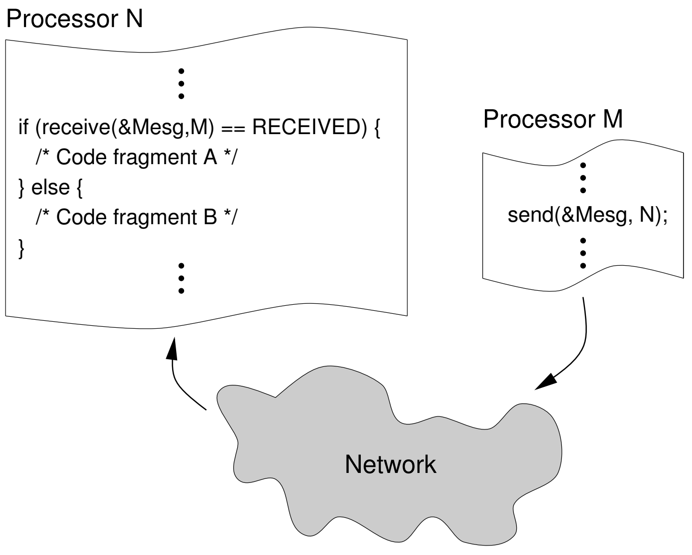
Trap-driven simulation is very similar to active memory but optimises the common case to its logical extreme: the no-action cases are not traced at all. To do so, trap-driven simulators exploit, or rather *abuse*, the available hardware facilities, such as the error correcting code (ECC) bits of memory elements, to store the status bits of the memory blocks. In this approach, the references which do not require any action (i.e. pointing to a valid memory block) can run at full hardware speed. Subsequently, the references to an invalid memory block cause a memory exception which invokes the simulator. To allow such control, the trap-driven simulator needs to be located in the OS (e.g. in the virtual memory system). Although this simulation technique is highly efficient, it clearly lacks the portability and generality of the previously mentioned simulation techniques. For example, the host machine's real cache(s) may interfere with the state mechanism, leading to inconsistent memory block states [137].

2.1.4 Tracing of parallel applications

Trace-driven simulation must be used with extreme care when studying the performance of parallel platforms. The control flow in parallel applications may be dependent on the outcome of *global events*, such as shared-memory accesses in multiprocessors or message passing communication. As these global events, on their turn, are affected by the latencies of the underlying hardware, the application's control flow may be non-deterministic. As a consequence, such non-deterministic execution behaviour can change the traces for different application runs and, evidently, for different parallel architectures [46, 53, 34]. So, for example, an address trace of an application generated on one specific multiprocessor platform is not by definition valid for another multiprocessor machine.

To illustrate this phenomenon, which is called the *global trace problem* [53], consider Figure 2.6. In this picture, the fragment of code for processor N performs a non-blocking receive operation (being a global event), which checks whether or not a message has arrived from processor M. Dependent on the result, the code follows a different execution path. If we assume that for a certain multicomputer architecture the message has not yet arrived, then the code segment *B* is executed and, as a consequence, this segment is also traced. However, when improving, for example, the interconnect of the multicomputer, the message may well have been arrived. This would invalidate the previously generated trace of code segment *B* since code segment *A* is now executed and must be traced accordingly.

The separation of the trace generation and trace processing phases in trace-driven simulation, which is often considered to be an advantage, is in this case the underlying problem which makes trace-driven simulation unsuitable for accurately modelling parallel architectures. As the trace generator executes in isolation, only affected by the latencies of the host computer it is running on and not by the target architecture's latencies, it is unable



```
if (receive(&Mesg,M) == RECEIVED) {
   /* Code fragment A */
} else {
   /* Code fragment B */
}
```

```
send(&Mesg, N);
```

Figure 2.6: A message-passing code fragment which is prone to the global trace problem.

to adequately represent the behaviour of the (possibly non-deterministic) applications executed on the target architecture. For this reason, most parallel architecture simulators apply *execution-driven simulation* rather than trace-driven simulation to guarantee the validity of the simulated events.

2.2 Execution-driven simulation

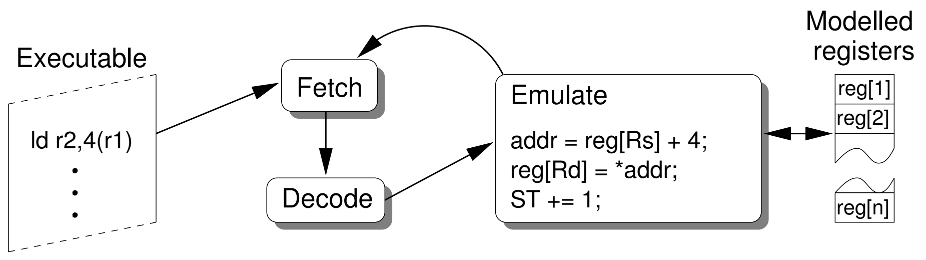
In execution-driven simulation, like its name already suggests, the execution of the application and the architectural simulation are interwoven. More specifically, the execution of the application, which generates the events for the simulation, is controlled by the simulator. The simulator may, for instance, control the application's execution by determining the execution path it has to follow. Because the simulator knows the timing consequences of the different events (e.g. the latency of a memory reference), it can "help" the application to execute as it would have on the target architecture. For this reason, execution-driven simulation is suitable for the simulation of parallel platforms. As the execution is dependent on the architectural simulation, non-deterministic application behaviour is automatically simulated as well. We should note that some authors directly relate the term execution-driven simulation to, what we call, *direct execution*. In this thesis, execution-driven simulation refers to a much broader class of simulation techniques. Direct execution will be discussed later in this section.

Since the execution of the application and the simulation are interwoven, the generated simulation events are consumed on-the-fly. Thus, the problem of storing large traces is non-existing in execution-driven simulation. On the other hand, the application has to be re-executed for every simulation run. It is therefore essential that the application's execution and the simulation are efficiently integrated. In the remainder of this section, we will show how several widely-used execution-driven simulation techniques address this efficiency issue.

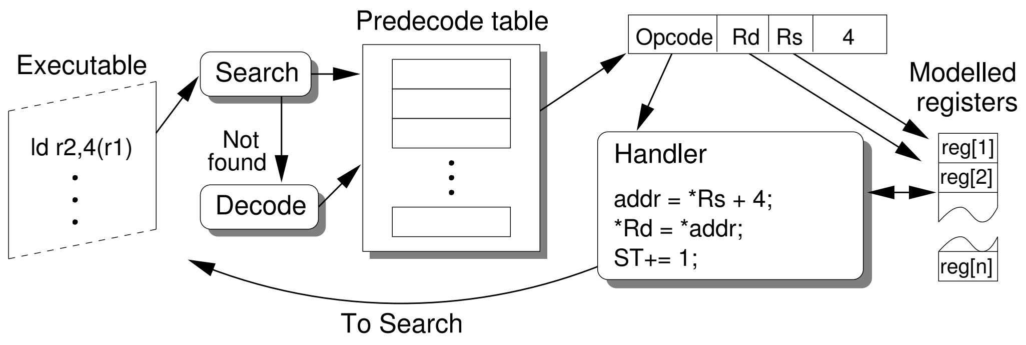
2.2.1 Instruction-level simulation

A conventional method to establish execution-driven simulation is to *interpret* the machine instructions of executables, which are compiled for the target architecture, and to *emulate* these instructions like they would have executed on the target machine. The most straightforward implementation of such an *instruction-level simulator* is to simply fetch, decode and execute each machine instruction like a real processor does [32]. Thus, an instruction is fetched by reading the contents of the executable's text segment. Subsequently, the instruction is decoded by determining its opcode and its operands. Finally, it is emulated by updating the machine state, such as the modelled register set. Figure 2.7(a) illustrates this type of simulation. In this figure, the variable `ST` (Simulated Time) keeps track of the virtual simulation clock. For the sake of simplicity, we assume that the load instruction only takes 1 cycle to execute. In reality, the simulator would, of course, model the latency more accurately by taking, for example, cache hits and misses into account.

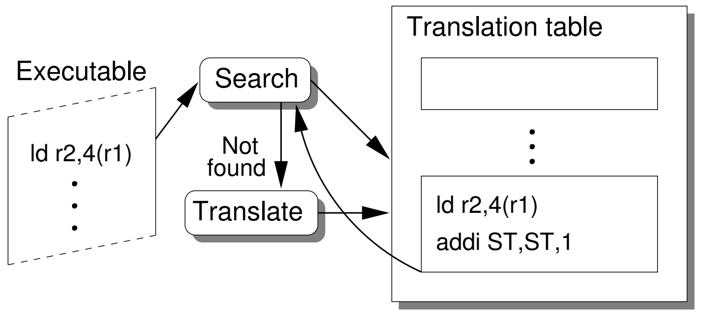
The major drawback of this simulation approach is its inefficiency. Therefore, several researchers have proposed and implemented different kinds of optimisations for instruction-level simulators. Two well-known optimisations are: *instruction predecoding* [9, 77] and



(a) Traditional instruction interpretation



(b) Dynamic instruction predecoding



(c) Dynamic translation

Figure 2.7: Several instruction-level simulation techniques. Figure (a) illustrates a conventional interpreting simulator, figure (b) a simulator with dynamic predecoding and figure (c) a dynamic translation simulator. In the figures, `ST` stands for Simulation Time, which is a variable keeping track of the virtual time.

translation [25]. With instruction predecoding, the cost of repeatedly decoding the machine instructions is avoided by first translating the executable to an intermediate, predecoded format which can be rapidly handled by the emulation engine. Alternatively, the predecoding of instructions can also be performed lazily while the program is emulated. In this case, the instructions which are executed for the first time are decoded in the traditional manner after which the results are cached in a predecode table. Any future references to the same instruction can then be handled quickly from this table. In Figure 2.7(b), simulation with dynamic predecoding is illustrated.

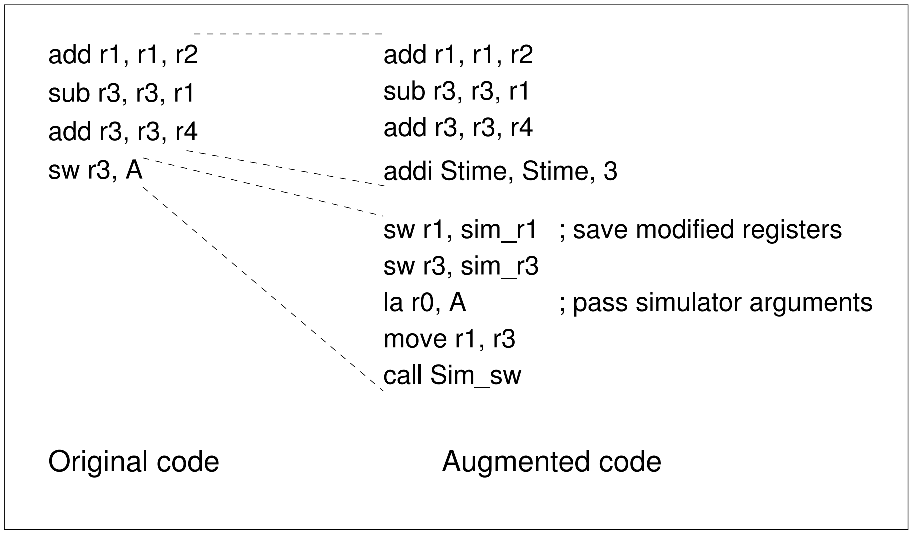
In translation, even more work is saved for the instruction decoder. By translating each target instruction directly into the corresponding host code which is used for emulating the instruction, the decoding stage has more or less been discarded. Like predecoding, translation can also be performed either statically or dynamically by means of a translation cache. In static translation, the target application is directly rewritten (i.e. cross-compiled) to the corresponding host emulation code. Similarly, dynamic translation translates any newly encountered target instruction on-the-fly, caches the generated code sequence and jumps to it. Hence, subsequent references to the target instruction can immediately jump to the cached code sequence. Compared to static translation, dynamic translation is more complex to implement but, unlike static translation, it allows to be used with, for instance, dynamically-linked code. The concept of dynamic translation is shown in Figure 2.7(c). In this example, it is assumed that the target and host instruction sets are identical.

Besides the optimisations mentioned above, there are quite a few additional optimisations, which will not be discussed, in order to reduce the time required for interpreting the instructions in instruction-level simulation. An alternative way to achieve higher simulation efficiency is by increasing the abstraction level at which is simulated. This brings us to execution-driven simulation techniques which do not interpret instructions. One of these techniques in particular, called *direct execution*, has become increasingly popular.

2.2.2 Direct execution

Direct execution [27] is a special case of static translation and is typically applied when simulating parallel architectures. This technique depends on two requirements. First, the host and target instruction sets should be identical. Naturally, such a requirement significantly limits the generality of this simulation technique. Second, in direct execution it assumed that explicit simulation of every instruction is not needed for obtaining a reasonably accurate performance estimate. To help to understand this, we will first explain how direct execution works in the context of the simulation of parallel architectures. In direct execution, two types of instructions are distinguished: *local* and *global* instructions. An instruction is local if its execution affects only the local processor. For example, all register-to-register instructions are local. Global instructions, such as shared memory accesses and network communication, may influence the execution behaviour of more than one processor. Evidently, global instructions and global events, which were introduced in Section 2.1.4, are one and the same.

The concept of direct execution is to execute the local instructions directly on the host computer (which is possible due to the equivalence of the host and target instruction sets) and to augment the code with cycle-counting instructions estimating the execution time of these local instructions. Thus, the performance impact of the local instructions is estimated statically, at compile-time. Subsequently, any encountered global instruction is trapped and explicitly simulated according to the specification of the target architecture. The idea to only simulate the global instructions stems from the belief that this type of instructions is often the primary concern of parallel computer architects: the focus of performance studies is typically on the network and memory design of parallel architectures. In these situations, the local computations are only relevant with respect to how they impact the timing of



```
add r1, r1, r2
sub r3, r3, r1
add r3, r3, r4
sw r3, A
```

```
add r1, r1, r2
sub r3, r3, r1
add r3, r3, r4
addi Stime, Stime, 3

sw r1, sim_r1   ; save modified registers
sw r3, sim_r3
la r0, A        ; pass simulator arguments
move r1, r3
call Sim_sw
```

Figure 2.8: An example of code augmentation for direct execution.

global instructions and do therefore not need to be simulated in detail. Hence, by applying simulation only where it is required (the other parts of the code, i.e. the local instructions, are directly executed), the simulation overhead is reduced considerably. As a consequence, high simulation efficiency is obtained at the cost of a small decrease in accuracy due to the static performance estimation of the local instructions.

Figure 2.8 uses a small fragment of MIPS assembly to demonstrate the code augmentation necessary for direct execution simulation. In this example, we assume that the target machine is a shared-memory multiprocessor. Moreover, the store instruction (`sw`) is considered to be a global event as it stores a value to shared memory. The effects of this instruction should therefore be simulated in detail. The other instructions, being local instructions, do not require to be simulated. As can be seen in the augmented version of the code, the two `add` instructions and the `sub` instruction are directly executed. Furthermore, the variable `Stime` is immediately updated to keep track of the execution time of the local instructions. So, here we estimate that simple arithmetic instructions take one cycle to execute. For the store instruction, `Stime` is updated by the simulator. Passing arguments to the simulator function `Sim_sw` is done via `r0` (the address) and `r1` (the value).

Though direct execution is fast, it does not allow the same degree of freedom in the evaluation of (parallel) architectures like the instruction-level simulators do. As was mentioned earlier, direct execution requires strong similarities between the instruction sets of the host and target platforms thereby constraining the general applicability of the technique. Moreover, because the performance of local instructions is determined statically, direct execution simulators are often unable to evaluate the architecture components which might affect the performance of the local instructions. For example, the performance evaluation of instruction caches or private data caches cannot, or only marginally, be performed by means of direct execution.

2.3 Simulation of parallel architectures

In the remainder of this thesis, the focus will be on the simulation of parallel computer architectures. More specifically, we propose a new computer architecture simulation methodology which we have embedded into an environment, or rather a *workbench*, that allows the performance evaluation of parallel platforms, and in particular, multicomputer platforms. At this point, the question may arise whether or not a new simulation environment, or even a new simulation methodology, is really needed. This because there already exists a fairly extensive collection of parallel machine simulators. Before motivating our work, we first present a brief overview of several state-of-the-art parallel machine simulators. Thereafter, we provide a discussion indicating where there is, in our opinion, room for improvement and in which we motivate our decision to develop a new simulation environment.

2.3.1 Related work

Parallel architecture simulators can be subdivided into three categories. First, the simulators that only address the simulation of shared-memory multiprocessors. Examples of these simulators are Tango [33], FAST [16], SimOS [119, 118], Wisconsin Wind Tunnel (WWT) [117, 96], Proteus [18] and SPAM [63]. All of these simulators apply direct execution to obtain tolerable simulation efficiency. SimOS also provides multiple levels of simulation. This enables the architect to position the simulation at an interesting state using a fast and abstract level of simulation. Thereafter, the interesting section is studied using a more accurate, and thus less efficient, mode of simulation. The WWT is probably the most extreme example of how the performance of architectural simulation can be boosted. It implements direct execution by means of trap-driven simulation: the ECC bits of the memory are set to cause an exception at a shared-memory reference which subsequently invokes the simulator. Furthermore, the WWT also exploits the inherent parallelism found in simulations of parallel architectures by executing the simulation in parallel, which is commonly referred to as *distributed simulation*. In Chapter 4, which includes an overview of the performance of several parallel architecture simulators, it is shown how WWT's optimisations are translated into performance.

The second type of simulator allows the simulation of both multiprocessors and multicomputers. The Rice Parallel Processing Testbed (RPPT) [28], SPASM [126] and SMART [105] are examples from this category. The RPPT and SPASM are, again, based on direct execution. Interesting about these two simulators is that they both use the simulation language CSIM [29] to implement their architecture models rather than using a traditional general-purpose language like most simulators do. As simulation languages often operate at a higher level than general-purpose languages, using a simulation language may result in a minor decrease of simulation performance but it may significantly improve the flexibility of the simulator. That is, it allows the modeller to focus on the *modelling* of the computer architecture rather than having to think about simulation details as well (these are taken care of by the run-time system of the simulation language).

The SMART simulator from Petrini et al. [105] does not use direct execution but applies some sort of static translation. As will be shown in the next chapter, there are quite some similarities between the techniques we have used in our simulation environment and the ones applied in SMART. We should note that our simulation environment was developed entirely independent of SMART and at approximately the same time. There are, however, some essential differences between the two simulation environments which will also be indicated in the next chapter.

Finally, simulators in the last category entirely focus on the simulation of multicomputer architectures. A well-known example is Talisman [9], which is an efficient instruction-level simulator using dynamic translation. This simulator is highly geared towards the simulation of only one multicomputer in specific, namely the Meerkat machine [8].

2.3.2 Discussion

Like the simulator overview of the previous section already suggests, we found that the majority of the parallel architecture simulators are multiprocessor simulators. This observation is based on the number of available simulators that evaluate the performance of the entire parallel platform. We did not consider the simulators that only simulate a multicomputer/multiprocessor network in isolation, such as MultiSim [88] and HSIM [55]. So, for starters, we think that the complete simulation of multicomputer architectures deserves to receive additional attention.

Moreover, it is obvious that most parallel architecture simulators use direct execution as the core simulation technique. As these simulators only simulate the global events (shared-memory references or explicit message passing), they emphasise the modelling of the communication and/or shared-memory system of the parallel architecture. This is, of course, justifiable for many performance studies. When studying, for instance, cache-coherent multiprocessors, of which the performance is often constrained by the coherency protocol, focusing on the memory sub-system definitely is a good approach. However, we question whether the emphasis should always be entirely on the communication/shared-memory system. For example, the performance gap between the CPU and the network in MPP systems has been reduced in the last decade. The performance of processors did improve by roughly a factor 20 in this period (compare, for instance, a 1988 MIPS R3000 with a 1998 DEC Alpha), whereas the latency and bandwidth of MPP networks improved by approximately two orders of magnitude (compare a 1988 Intel iPSC/2 with a 1998 Cray T3E) [125, 52]. So, although the network latency is still a dominant factor in the overall performance of many parallel applications, this problem has become less severe. Moreover, latency hiding techniques, such as multithreading and data prefetching [48], may further decrease the impact of communication overhead on the application performance. We therefore believe that it becomes increasingly important to include, when necessary, the explicit simulation of the performance behaviour at the local nodes rather than only simulating global events. Hence, in this respect, we think that direct execution (which only allows the simulation of global events) is not a suitable simulation technique.

Another potential drawback of the previously mentioned simulators is that most of them apply “low-level” techniques, such as direct execution or dynamic translation, in order to yield high simulation performance. This implies that these simulators are highly dependent on both the host and target processors’ instruction set architectures. Again, this is no problem when the focus is on the communication/shared-memory system of the architecture. However, when the scope of the evaluation study is broader, including for instance the type of processor on a node, these architecture dependencies might seriously hamper the flexibility of the simulator.

From the above can be concluded that the existing simulators typically trade performance for flexibility. The simulation techniques they use are efficient but are also limited in their capabilities (e.g. direct execution) or highly architecture dependent. This is, in our opinion, a major weakness of current simulator technology. We think it is essential for simulators to provide a high level of flexibility in order to guarantee their applicability for a wide range of performance evaluation studies. As we prefer the modelling to be more flexible, this means that we have to find alternative ways to improve the simulation performance. In other words, we need a simulation method that allows for the performance evaluation of parallel computer architectures in a flexible manner and which is also competitive performance-wise with, for instance, direct-execution simulation. We use the term *flexible* in the sense that the simulation method should allow for simulating both local and global instructions and should also be highly architecture independent. These flexibility requirements facilitate the evaluation of all aspects of the parallel machine, including for instance the type of processors.

The need for a flexible simulation environment is illustrated by the case studies presented in the last chapters of this thesis. The focus in each of these case studies is on different aspects of the computer architecture. Without a flexible simulation method, we would not have been able to perform these studies using only one simulation framework.

The next chapter describes the Mermaid simulation environment, in which we have implemented our ideas regarding flexibility and performance. The primary focus of the Mermaid environment is on the performance modelling of multicomputer architectures. However, it is not restricted to this type of platforms. Support for the evaluation of different types of multiprocessor platforms (e.g. UMA, NUMA, etc.) is also provided. To establish a high degree of flexibility while providing high simulation performance, Mermaid applies a simulation method that is a combination of trace-driven and execution-driven simulation. This will be explained elaborately in the next chapter.

Chapter 3

Mermaid[†]

"There are finer fish in the sea than have ever been caught."
Irish proverb

To evaluate the performance of MIMD multicomputer architectures, we developed the Mermaid (Modelling and Evaluation Research in MIMD ArchItecture Design) simulation environment. The design of this *architecture workbench* was guided by a number of requirements. First of all, the architectural simulation must be efficient. This implies that it should be possible to simulate representative application loads within reasonable time. In this context, the term *reasonable* is hard to define. We will use other efficient multiprocessor/multicomputer simulators to act as a reference for Mermaid's simulation performance.

Like was shown in the previous chapter, most modern parallel architecture simulators address simulation efficiency quite aggressively and, by doing so, typically trade the gained performance for flexibility. A good example are the commonly used direct execution simulators which are dependent on the host's instruction set and which limit the architectural evaluation to the parts that are really simulated (i.e. the global events). Flexibility is our second requirement, which means that Mermaid should yield good performance while providing a high degree of modelling freedom. It supports, for example, the evaluation of a wide range of design options by means of parameterisation and configuration of the simulation models at different levels: from processor parameters, such as cache specifics, to switching and routing techniques in a message-passing communication network. Moreover, our architecture models are highly modular. As a result, the changing of important simulation-model components, like the type of processors, requires only little remodelling effort. To allow this high degree of modelling freedom, Mermaid does not apply simulation techniques which inherently limit flexibility, like direct execution does. Instead, we use a method which is a combination of trace-driven and execution-driven simulation. This simulation methodology will be described in detail in the next section.

Finally, good simulation accuracy is our third and last requirement. Like it is hard to determine how well the absolute simulation performance is, the same is true for accuracy. Nevertheless, we aim at modelling errors that on the average do not exceed 5%. This should be accurate enough to perform design space exploration. Here, we define *modelling error*

[†]This chapter is based on [108, 109, 114].

as the difference between the predicted execution time of an application and the actual execution time obtained by the real machine.

Like it was already shown for the simulation efficiency and flexibility, the aforementioned requirements typically conflict with each other. For example, achieving higher simulation efficiency may result in lower accuracy and less flexibility. In Mermaid, we have therefore tried to find a good tradeoff between these factors. To do so, we address the issue of simulation performance in a different way compared to most other parallel architecture simulators. Instead of applying low-level and architecture-dependent simulation techniques (such as direct execution), we *exploit* the tradeoff between abstraction level (i.e. level of detail) and performance allowing us to zoom in on only the architecture specifics which are interesting to us. As we will show, this approach results in flexible and efficient simulation of parallel computer architectures and is still capable of obtaining good simulation accuracy.

3.1 The simulation methodology

The simulation of a computer architecture can be regarded as a process which involves two tasks. In the first place, the behaviour of the computer architecture has to be captured in a simulation model. To use this model for experimentation, it should be driven by a *workload* representing the behaviour of an application. So, the second task consists of describing the application behaviour in one way or another (e.g. by using real, compiled programs) with the intention to create a workload for the architecture simulation model. We refer to this as *application modelling*.

In Mermaid, the application behaviour is described in terms which are unrelated to architecture specifics while the architectural behaviour is described in terms of its components and its interactions. By *separating* the modelling of application behaviour and architecture behaviour, a flexible evaluation of either one is possible without the other being affected. This decoupling of application and architecture modelling is a key feature of Mermaid. It allows, for example, the modelling of application behaviour at different levels of abstraction without the architecture models being affected. This will be explained in more detail later in this chapter.

To include both application and architecture behaviour in one simulation, an *intermediate layer* is required that bridges the gap between them and which strives for efficient and optimal utilisation of hardware resources. In other words, this intermediate layer maps one complex system (the application) to another complex system (the computer architecture) [129]. To do so, it must define the interface between the application and architecture levels. In Mermaid, this interface is established by means of traces of events, called *operations*. These operations represent the processor activity, memory I/O and message-passing communication due to the execution of an application. Here, one should think of events such as “add two integers”, “load a word from memory” or “send a message from one processor to another”. The actual form of the operations will be discussed later on.

The intermediate layer translates the application behaviour into an appropriate trace of operations which is subsequently consumed by the simulator at the architecture level. The interface between the intermediate layer and the architecture level is bi-directional, allowing the architecture level to give *feedback* to the intermediate layer. As will be explained later, this may be necessary for the correct generation of operations. The concept of operations,

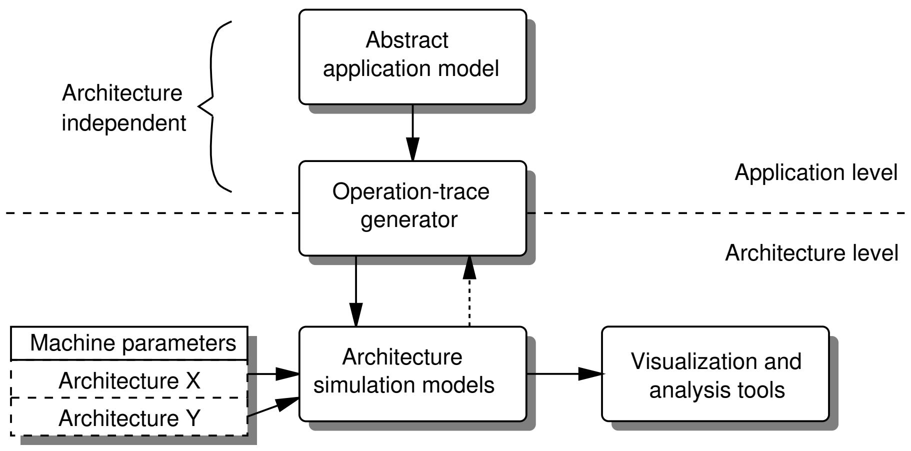

Architecture independent
Abstract application model
Operation-trace generator
Application level
Architecture level
Machine parameters
Architecture X
Architecture Y
Architecture simulation models
Visualization and analysis tools

Figure 3.1: Mermaid simulation environment.

which essentially forms the heart of our simulation methodology, results in a technique we call *operation-driven simulation*.

3.1.1 The simulation environment

The separation of application and architecture is reflected by Mermaid's simulation environment. Figure 3.1 depicts the simulation environment. It is layered in a natural fashion and shows a clear distinction between the application and the architecture levels. We will first present a brief description of each of these two levels, after which the remainder of this chapter gives a more comprehensive overview of the design and implementation issues regarding the various simulation environment components.

The application level

The application level consists of an abstract application model which specifies workloads in an architecture-independent manner. These workload specifications are either the sources of real programs that have been instrumented with *annotations* describing the exact execution behaviour or they consist of *stochastic representations* of application behaviour. Note that we explicitly use the term *specifications* since the workloads do not require to be executable at this level. If we take our code annotation method as an example, the workload specification consists of a C source program that has been instrumented with calls (i.e. the annotations) to a library which generate the operations whenever the application is executed. So, the actual workload description, being the C code, is *not* directly executable. Furthermore, the workloads may range from full-blown parallel programs to small benchmarks used to tune and validate the (machine parameters of the) simulation models.

The architecture level

The operation-driven architecture simulation models reside at the architecture level. These models are implemented in a modular fashion and are highly parameterised. By adjusting the machine parameters, which are stored in a separate data-base (see Figure 3.1), different architecture configurations can swiftly be evaluated. Like the RPPT [28] and SPASM [126] simulation environments, our architecture models are also written in a special-purpose (simulation) language rather than in a traditional general-purpose language. This considerably enhances the flexibility of the architectural simulator as it reduces the time to develop or adapt the architecture models.

At the architecture level, Mermaid also provides a suite of tools in order to visualise and analyse the simulation output. Visualisation of simulation data can be performed at run-time or post-mortem. Moreover, a tool called RAPID [110] facilitates the statistical analysis of the simulation output. This tool allows the user to select interesting information and to perform a range of standard statistical methods on the selected data.

3.1.2 Operation-driven simulation

The operation-driven simulation technique can be regarded as a combination of trace-driven and execution-driven simulation. In this technique, a *trace generator* implements the intermediate level between the application and architecture levels. It provides the architectural simulator with the operation-traces which account for the overall application behaviour, thus including both local and global (i.e. affecting the execution behaviour of other processors) instructions.

The trace generator produces a separate operation-trace for each processor within the multicomputer architecture model according to the workload specifications at the application level. For this purpose, it mimics concurrent execution by means of threads [11]. Each thread accounts for the behaviour of one processor (or node) within the parallel machine. In this respect, we assume that there is only one user process running on a processor. So, we do not consider issues like process scheduling. To guarantee the validity of the traces and solve the global trace problem, which was discussed in Section 2.1.4, a form of execution-driven simulation is established by applying *physical-time interleaving* [34, 98]. In this technique, the trace generation is interleaved with the simulation of the target architecture and the traces are generated on-the-fly. This allows the architecture simulation to control the trace generator by giving it *feedback* with respect to the scheduling of global events. More specifically, whenever a thread encounters a global event, it is suspended until it is explicitly resumed by the simulator (depicted by the broken arrows in Figures 3.1). Subsequently, the simulation does not resume a thread until all other threads have reached the same point in simulated time as the suspended thread. When this has happened, no other events can affect the global event within the suspended thread anymore. Therefore, the suspended thread can be safely resumed again. Such a *thread-scheduling* scheme, under the control of the simulator, guarantees that the multicomputer trace is exactly the one that would be observed if the application was actually executed on the target machine.

In order to illustrate the thread-scheduling scheme, consider the example of the global trace problem from Section 2.1.4 again: a thread requests whether or not a certain message has arrived (i.e. it performs a non-blocking receive) and dependent on this result it follows a different execution path. In Figure 3.2a, this situation is depicted. A “box” in between the trace generator and simulator denotes an operation-trace produced by a single thread. At the moment the non-blocking receive operation is issued by the trace generator, the receiving thread is suspended. The operations from all the other threads, however, continue to be simulated up to the moment in simulated time at which the requesting thread was suspended.

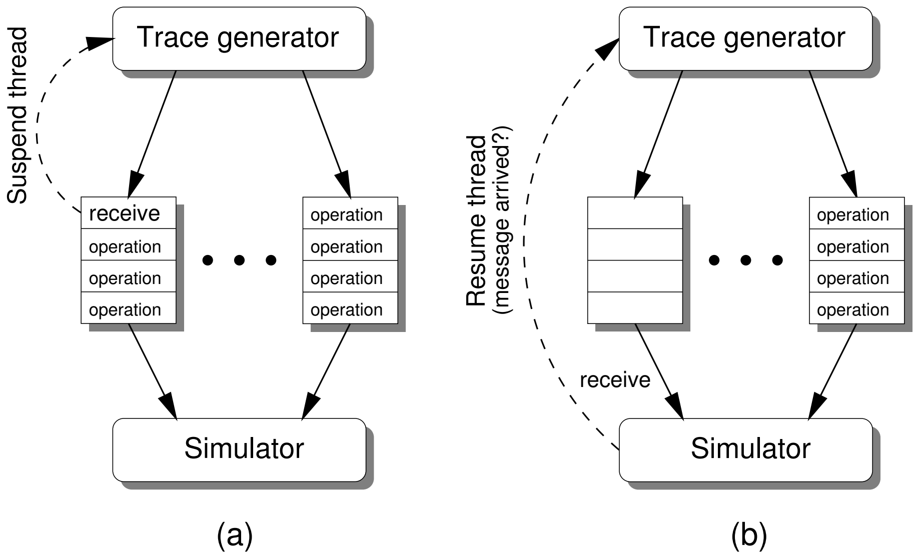


Figure 3.2: Thread scheduling under control of the simulator to guarantee the validity of the multiprocessor traces.

Because all message passing has now been simulated up to the moment of the request, the simulator can safely resume the requesting thread by giving it the required information on the presence of the message, as illustrated in Figure 3.2b.

Application-level synchronisation

Synchronising the trace generator with feedback from the simulator may affect the trace generating performance, and thus the overall simulation performance, as the trace generation is constrained by the simulator's efficiency. However, in many cases, a short-cut optimisation for synchronising the trace-generating threads can be applied. A lot of multicomputer applications, and especially the ones belonging to the popular class of SPMD (Single-Program, Multiple-Data) programs, contain fixed communication patterns. In other words, their communication does not depend on the underlying architecture which makes them to behave deterministically. This type of application allows for synchronising the trace-generating threads at the application level rather than synchronising them with simulator feedback. As a result, the simulation does not need to be execution-driven. Instead, the architectural simulator can now operate in pure trace-driven mode, thereby not constraining the execution of the trace generator.

The synchronisation at the application level is illustrated in Figure 3.3. Again, a thread is suspended when executing a global event, such as a blocking receive. In this case, however, the suspended thread can be resumed directly by the thread which produces the requested data. Whenever the latter thread is ready, it copies the required data to the suspended thread's local memory and subsequently wakes it up. Because data is directly exchanged between threads at the application level, it is not necessary to simulate the explicit data transfer at the architecture level. Instead, if a processor sends a message of size N to another processor, the simulator models this by sending an "empty" message of size N.

We should note that if the execution of the application does not depend on global events,

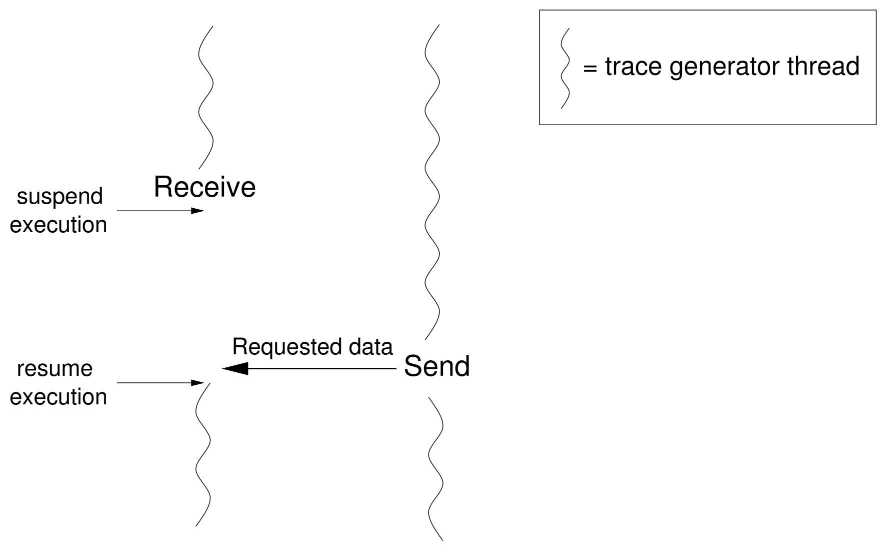


Figure 3.3: Application-level synchronisation between trace-generating threads.

then it is neither needed to synchronise nor to copy data between trace-generating threads because the threads' execution does not depend on the data which is transferred between processors. Hence, in this situation, the simulation does not require data consistency.

3.1.3 Computation versus communication

Many applications running on distributed memory multicomputer platforms, and especially the extensive class of scientific applications, contain coarse-grained computations alternated with periods of communication. Typically, these computation and communication phases are distinct. Therefore, Mermaid separates the simulation of application behaviour into a *computational model* and a *communication model*. This is depicted in Figure 3.4. The models operate at a different level of detail and define their own set of operations. The computational model simulates the application's computational behaviour. It models the incoming computational operations at a level of *abstract machine instructions*. Communication operations are not simulated by this model, but are directly forwarded to the communication model. Subsequently, the communication model accounts for the application's communication behaviour. To address the issues of synchronisation and load-balancing properly, the communication model simulates the computational delays found in between communication requests at the task level. A parallel workload for this model therefore resembles a graph containing computational tasks and global events (communication operations). The computational tasks are derived from the computational model, which constructs them by measuring the simulated time between two consecutive communication operations.

As can be seen from Figure 3.4, the cooperation between the computational and communication models results in a two-stage meta model, which we call Mermaid's *hybrid model*. This hybrid model allows for simulation at different abstraction levels. If accuracy is required, then the complete hybrid model (both the computational and communication models) can be used. However, if there is only the need for a quick and less accurate analysis of the architecture performance, which is commonly referred to as *fast prototyping*, then just

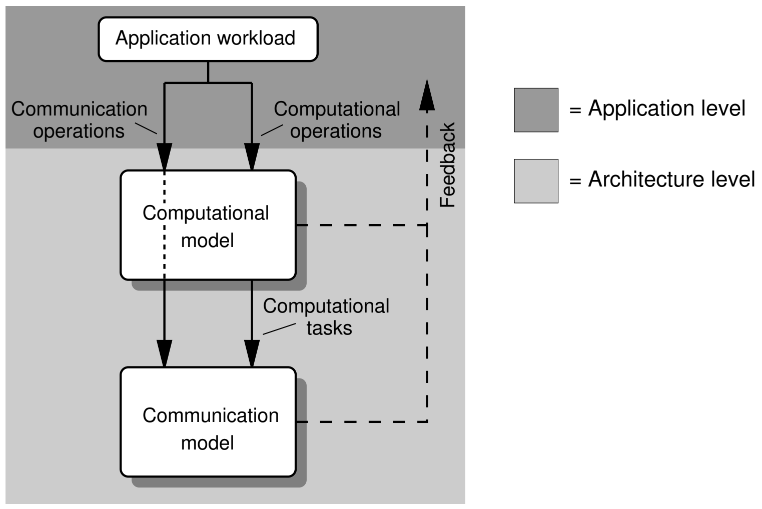


Figure 3.4: The computational and communication models.

using the communication model might be sufficient. In that case, the task-level operation traces must be directly produced by the trace generator. To some extent, this is similar to what happens in the direct execution simulation technique: the performance of the code segments in between two global events (forming the computational tasks) has to be statically estimated.

The idea of having multiple levels at which simulation can take place is also present in SimOS [119, 118]. However, Mermaid is more static in this respect: while SimOS allows the simulation to dynamically change from one level to another, Mermaid can only operate at one abstraction level during the whole simulation.

Computational operations

The computational operations are abstract machine instructions. They are based on a load-store architecture. This does not imply that other types of architectures cannot be simulated. The current set of computational operations can easily be extended or used as a building block for more powerful operations in order to support the modelling of alternative types of architectures.

Table 3.1 shows the computational operations. The first category consists of operations for transferring data between registers and the memory hierarchy. Parameters of these operations indicate the type of memory reference and, if appropriate, the memory location. As our focus is on distributed memory multicomputers, atomic operations to support the modelling of shared memory platforms are currently not provided. These atomic operations, like *read-modify-write*, can be added with relative ease.

The second category of operations are arithmetic functions that solely operate on registers. The associated parameter indicates the type on which the function should be performed. This can be integer, single precision or double precision floating point.

The third category of operations is associated with instruction fetching. With the *ifetch* operation, an instruction fetch from a memory location can be modelled. As the simulation does not interpret machine instructions, the simulator is not aware of loops and branches. The application trace generator evaluates loop and branch-conditions, and produces the op-

Description	Computational operations

Accessing memory	*load(mem-type, address)*
	store(mem-type, address)
	load([f]constant)

Performing arithmetic	*add(type)* *sub(type)*
	mul(type) *div(type)*
	...

Instruction fetching	*ifetch(address)*
	branch(address) *syscall()*
	call(address) *ret(address)*

Table 3.1: Computational operations.

eration trace for the invocated control flow. This implies that every invocation of a loop body is individually traced and leads to recurring addresses of instruction fetches.

The *branch* operation models the target processor's branching latency and triggers its branch prediction scheme, if available. Finally, the *syscall*, *call* and *ret* operations mimic the overhead involved with system calls or function calling.

Communication operations

The operations that act as input for the communication model are mainly based on straightforward message passing. The format of these operations, as shown in Table 3.2, is similar to the simulation events used by HSIM [55], which is a trace-driven network simulator of hypercube-based multicomputers. Both synchronous (blocking) and asynchronous (non-blocking) communication operations are supported. Furthermore, it is assumed that nodes are uniquely numbered within a given communication network.

Note that the *send* and *asend* operations do not include the data that is transferred. The actual data transfer (not the synchronisations) between the trace-generating threads is always performed at the application level, as was illustrated in Section 3.1.2. Copying data directly at the application level rather than performing the data transfer at the lower, architecture level is, of course, more efficient. It does not suffer from the redundant overhead of copying data between the application and architecture levels. If application-level synchronisation is used (i.e. the communication patterns are fixed), then the trace-generating

Description	Communication operations

Synchronous communication	*send(message-size, destination)*
	recv(source)

Asynchronous communication	*asend(message-size, destination)*
	arecv(source)

Computation	*compute(duration)*

Table 3.2: Communication operations.

threads directly copy the data to the correct destination address within the receiving thread. But, in the case simulator feedback is required, it may be architecture dependent which data is actually received by the destination processor. There could, for example, be multiple processors sending a message to the receiving processor. Therefore, the access to the copied data is facilitated by an identifier which is returned by the *recv* and *arecv* operations. This identifier, pointing to a data structure containing the appropriate message information, is determined by the corresponding *send* or *asend* operation.

Computation performed within the communication model is simulated at task level by means of the *compute* operation. This operation simply tells a processor to be busy for a certain duration.

3.1.4 The implications of operation-driven simulation

Applying operation-driven simulation has several consequences. As the operations abstract from the processors' instruction sets, the simulators do not have to be adapted each time a processor with a different instruction set is simulated. We simulated, for example, both multicomputers based on Inmos Transputers [58] and on Motorola PowerPC [94] processors with little remodelling effort. Essentially, operation-driven simulation shifts a part of the required modelling effort from the architecture level to the application level: architectural modelling becomes more flexible due to the abstract operation-interface, whereas at the application level, raw applications cannot be used as workloads anymore because the simulators are driven by operations (rather than interpreting real instructions). As a result, application behaviour must be modelled explicitly. Because our focus is on the evaluation at the architecture level, we definitely regard the gain of flexibility at the architecture level at the cost of a slightly more complex application level as an advantage. Furthermore, as will be shown later in this chapter, the explicit modelling of application behaviour opens new perspectives for simulating workloads at different levels of abstraction and accuracy.

Simulating at the level of operations rather than interpreting real instructions allows for modelling only the *timing consequences* of instruction execution. Most of the state transitions caused by instruction execution, such as the actual storing of a value in a register, do not need to be modelled. Therefore, it is not necessary to store large quantities of state information during simulation runs. For example, register and memory contents do not have to be modelled and simulated caches only need to hold addresses (tags), not data. As a result, our approach may yield higher simulation performance compared to the more traditional instruction-level simulation techniques.

On the other hand, the strength of abstraction is also Mermaid's weakness. The loss of information, such as the lack of register specifications in the operations, prohibits an *accurate* low-level simulation of, for example, the processor pipelines. This means that Mermaid is not suited for purposes like compiler testing or debugging.

3.1.5 Implementation issues of the operation-interface

The actual form of operations and the way in which they are transferred from the trace generator to the consuming architectural simulator may greatly influence the performance and flexibility of the entire simulation system. If the operations are defined in ASCII format, the simulator needs to perform quite a lot of parsing and decoding of the incoming operation

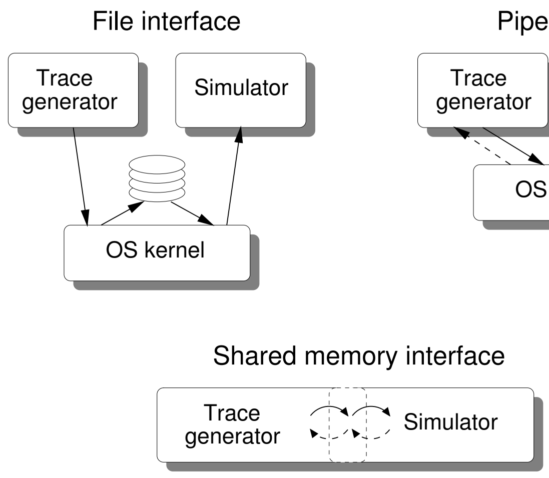


Figure 3.5: Different types of trace-interface implementations. The possibility of simulator feedback is illustrated by a broken arrow.

traces, thereby hampering the simulation efficiency. On the other hand, the ASCII operations may be useful for *direct* verification or validation of the application behaviour as they are readable. Alternatively, operations in (predecoded) binary format reduce the decoding overhead in the simulator and therefore yield higher simulation performance, but they are less friendly to study in isolation.

The interface between the trace generator and the simulator, transferring the operation-traces, can also be implemented in various ways. This is illustrated in Figure 3.5. A reasonably simple interface is obtained by Unix' pipes. This mechanism also allows to store the generated traces on disk (if their size allows this) in order to re-use them and thereby speeding up consecutive simulations of the same architecture. However, in this case, extreme care must be taken that the traces are only re-used for identical architectures to prevent the global trace problem (see Section 2.1.4). On the downside, the major disadvantage of pipes and files is their relatively poor performance as all data is copied through the OS kernel. An additional drawback of a file interface is that it is, unlike other types of interfaces, not suitable for providing simulator feedback.

A more efficient interface is obtained when performing communication via a piece of memory which is shared between the generator and the simulator. Although this method allows for a much faster interface than with pipes or files, it also requires more bookkeeping. For example, the generator and simulator must now synchronise to guarantee mutual exclusion with respect to the piece of shared memory.

The shared memory interface allows for different types of optimisations. An obvious one is to use two buffers in shared memory: a *primary* buffer containing the operations which are being consumed by the simulator and a *shadow* buffer which is filled by the trace generator concurrently with the consumption of the primary buffer. When the primary buffer is empty, the functionality of the two buffers is swapped.

As each of the aforementioned implementation approaches has its pros and cons, Mermaid includes them all by means of conditional compilation. By default, however, the simulators optimise for speed: binary operations and a shared memory interface between generator and simulator are used.

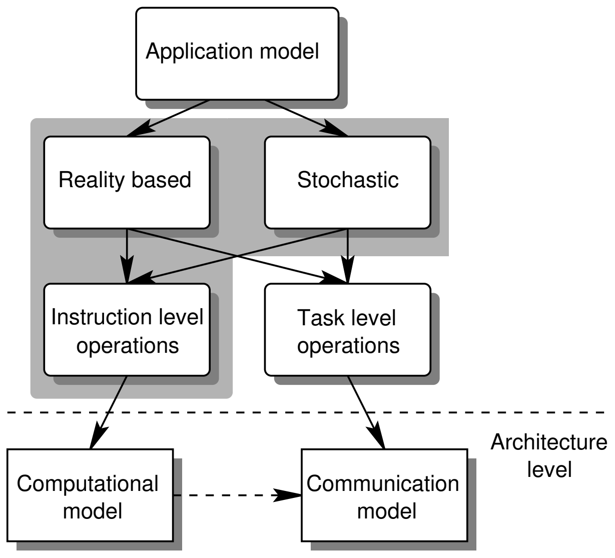


Figure 3.6: Application modelling within Mermaid. The large shaded area indicates the modelling path currently supported.

3.2 Workload modelling

In Figure 3.6, the workload modelling framework of Mermaid is illustrated. Application behaviour is modelled explicitly, enabling us to model a workload at various abstraction levels and with different degrees of accuracy. A workload is either based on a real application or it is synthetic and produced by some stochastic process. Furthermore, both real and synthetic workloads can model computation either at the level of abstract machine instructions or at the level of tasks. Computation at the instruction level is typically simulated by the computational model, whereas task level operations are simulated by the communication model.

Currently, Mermaid only supports the generation of abstract instruction-level operations, as depicted by the large shaded area in Figure 3.6. We will therefore limit our discussion to this area of workload modelling.

3.2.1 Reality-based workload modelling

To obtain a realistic model of application behaviour, it is required to trace a real program. Traditionally, the tracing of applications is often performed by augmenting either the assembler code [76, 135], the object code [132] or the executable itself [79, 78]. Of these three tracing techniques, instrumenting executables is the most convenient in its use as this technique does not require the source of the program to be available. However, all three techniques, and especially the the ones dealing with compiled machine code, introduce dependencies on either the host or the target architectures. As we strive for an architecture independent application level, this is undesirable. So, to guarantee a high degree of architecture independence at the application level, we decided to refrain from these techniques. Instead, Mermaid augments programs at the highest level possible: that of the high-level language itself. This means that the source code of applications, which in our case has to be written in C

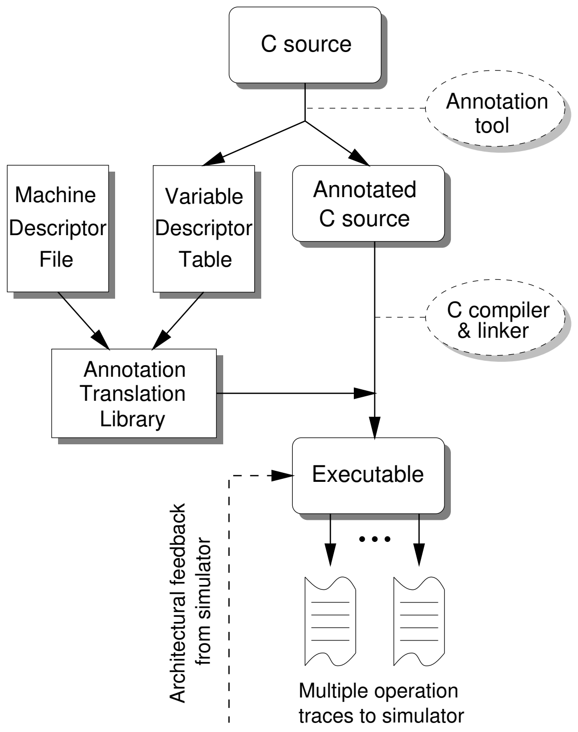

C source
Annotation tool
Machine Descriptor File
Variable Descriptor Table
Annotated C source
C compiler & linker
Annotation Translation Library
Executable
Architectural feedback from simulator
Multiple operation traces to simulator

Figure 3.7: Generation of operations by augmenting real applications.

[69], is instrumented with annotations following the program's memory, computational and communication behaviour. Hence, Mermaid's abstract application model for reality-based workloads consists of augmented high-level language programs. Because these augmented source codes are highly architecture independent, the instrumentation has to take place only once, after which the application model can be used to evaluate any multicomputer architecture.

Figure 3.7 gives an overview of how Mermaid generates operations by augmenting real applications. A special annotation tool instruments the C source code and constructs a so-called *variable descriptor table*. This table determines whether a variable is global, local, or a function argument. Additionally, it contains information on the addresses of variables, whether they are placed in a register or not and the types of the variables. The annotation tool also models several common and straightforward compiler optimisations by optimising (i.e. rewriting) the generated annotations. These optimisations include common subexpression elimination and constant folding [3].

The Annotation Translation Library (ATL) is the trace generating core, which is linked to the annotated application. As the annotations are simply calls to the ATL, the annotations are dynamically translated into the appropriate trace of operations when the obtained executable is executed. For this purpose, the ATL uses the variable descriptor table and a *machine descriptor table*. The latter contains architecture-dependent information necessary for the trace generation (e.g. instruction size, number of registers, etc.). When, for example, an annotation indicates that a variable should be loaded, the generator uses the information from both tables to translate the annotation into the appropriate instruction fetch and memory operations. The ATL can thus be regarded as a kind of generic compiler. It performs the translation of annotations according to the runtime model and addressing capabilities of

```
double a, x[N], y[N];

void foo(...) {

   int i;

   [...]

   for (i = 0; i < N; i++) {
      a += x[i] * y[i];
   }

   [...]
}
```

```
double a, x[N], y[N];

void foo(...) {
   int i;

   set_function(foo, (void *)foo);

   [...]

   for (assignIc(local(i), 0), i = 0;
        setPC(c1),
        arithIc(REG, N, local(i), CMP), i < N;
        arithIc(local(i), 1, local(i), ADD), i++) {

      a += x[i] * y[i];

      arithDF(REG, array(&x, local(i), &x[i]),
                   array(&y, local(i), &y[i]), MUL);
      arithDF(&a, REG, &a, ADD);
      setPC(l1);
   }
   setPC(l1);

   [...]

   ret_function();
}
```

Figure 3.8: A double precision inner-product (top) and its annotated version (bottom).

the target processor. In this approach, only the ATL and the machine descriptor file are architecture dependent and may need to be updated in order to generate operation-traces for a new (possibly non-existing) architecture.

To illustrate the process of instrumentation, consider Figure 3.8. The upper box shows a code fragment of a double precision inner-product (*ddot*), whereas the lower box shows the annotated version of the same code fragment. The annotations *set_function*, *ret_function* and *setPC* describe the control flow behaviour of the program. For example, the *setPC(label)* annotation is some sort of basic block indicator: the first time a *setPC* is triggered, it saves the current (pseudo) program counter together with its label. Subsequent calls to a *setPC* with an identical label will reset the program counter to the saved value. For example, the two *setPC(l1)* annotations at the end of the loop in Figure 3.8 take care that the program counter is correctly updated when the loop has finished. The first *setPC(l1)* saves the current program counter, which has to be adopted when the loop has finished. The second *setPC(l1)* (outside the loop), which is reached after the conditional expression of the for-loop has failed, sets the program counter to the saved value.

The $assign_{type}$ and $arith_{type}$ annotations describe the computational behaviour of the application. In these annotations, the type must be specified on which is operated. For instance, *Ic* means "Integer constant" and *DF* means "Double Float". If required, these annotations also instruct the ATL to model the fetching of the operands. To do so, special handles, such as *local* and *array*, are used to query the variable descriptor table to determine the location (e.g. in a register, on the heap, on the stack, etc.) of the operands. Additionally, the keyword *REG* explicitly specifies that the source or destination is a register.

For now, the performance impact of system calls to the OS is modelled by an annotation which simply generates the *syscall* operation. The *syscall* operation mimics the overhead involved with the system call. This method is not accurate enough for applications which contain a large number of system calls. In these cases, the semantics of the system calls might have to be modelled to generate a trace of operations which properly represents the system call's behaviour. Moreover, the OS *itself* is not modelled in Mermaid. To our knowledge, of all existing parallel architecture simulators, only SimOS [118, 119] is capable of doing this.

To generate the multiprocessor traces, the ATL models concurrent execution by means of threads. For this purpose, the tool which performs the instrumentation of C programs also provides some support for converting (single-threaded) target applications into multi-threaded programs. For SPMD applications, this conversion is fully automated. For other classes of applications, manual conversion to the threaded code is still necessary.

Modelling communication

The ATL is also capable of inserting annotations describing the message-passing behaviour of an application. These communication annotations directly map onto operations, such as the *send* listed in Table 3.2. As the associated parameters of these operations are based on the platform's physical topology, the modelling of communication still reflects some of the underlying hardware characteristics. Ideally, such architectural details are not visible at the application level. One way to achieve this, is by using application-defined communication structures, also called *virtual topologies* [120]. This requires an extra translation step that translates the virtual communication requests to physical communication requests. This translation must be guided by a pre-established mapping of the virtual communication topology on the platform's physical communication topology [120]. In the context of the Mermaid simulation environment, such a translation could be performed at the intermediate layer between the application and architecture levels, i.e. by the trace generator.

A more radical method to hide architectural details at the application level is to define the communication annotations such that they are based on a virtually shared memory model. This approach, which is also followed by the SMART simulation environment [105], is similar to that of the High Performance Fortran initiative [54]. As a result of this approach, all explicit communication at the application level is removed and substituted by memory references. This implies that a component of the simulation environment, presumably the trace generator, issues the communication requests rather than the application itself. Naturally, the generation of these communication requests is dependent on the distribution of data over the different processors. So, by manually specifying how the data is distributed, different data distributions can easily be evaluated without the need to change the applications.

There are, of course, several issues that have to be addressed when adopting a virtually shared memory model to describe application behaviour. Access to distributed data needs to be synchronised to avoid, for example, data hazards like RAW (*read after write*), WAR (*write after read*), and WAW (*write after write*). These hazards are similar to the data hazards in instruction pipelining [52].

Neither the virtual topologies nor the virtually shared memory model are implemented, both are future work.

3.2.2 Stochastic workload modelling

Besides the tracing of real applications, we have also investigated techniques to generate operation-traces from stochastic application descriptions. In such a scheme, the abstract application model consists of descriptions expressing the application behaviour using probabilities. Evidently, this technique represents the behaviour of (a class of) applications only with modest accuracy. However, it offers more flexibility than the tracing of real programs. For example, rapidly adjusting the application behaviour is fairly easy using this technique, which can be useful when fast-prototyping new architectures. At this moment, our stochastic application modelling framework only supports the modelling of computation for a uniprocessor system. For this reason, we limit the discussion in this section to sequential computing only.

The probabilistic descriptions are written in the language SEA (acronym for Stochastically Expressing Applications), which was especially designed for this purpose. A tool, called the Stochastic Trace Generator (STG), interprets the SEA descriptions and generates operation-traces from them. Like in the case of realistic workloads, the STG uses a machine descriptor file to tune the traces for a particular target architecture. For example, the descriptor file's information on the available number of registers in the target architecture is used to filter out memory accesses from the operation-trace in order to model register accesses.

Figure 3.9a shows the general structure of a SEA description. From a high-level point of view, an application description is constructed of a sequence of one or more *kernel* descriptions. These kernel descriptions consist of three parts: a data part, an operation part and a general part. The data section describes what kind of variables can be manipulated within a kernel. The operation section specifies the types of operations taking place within a kernel. Since this section may also contain sub-kernel descriptions, the complete application model is represented by a tree of kernel descriptions. Finally, the general section describes issues like the number of operations to be generated within a kernel, addresses of segments, etc. To reduce the size of SEA descriptions, inheritance is supported. This allows the specifications

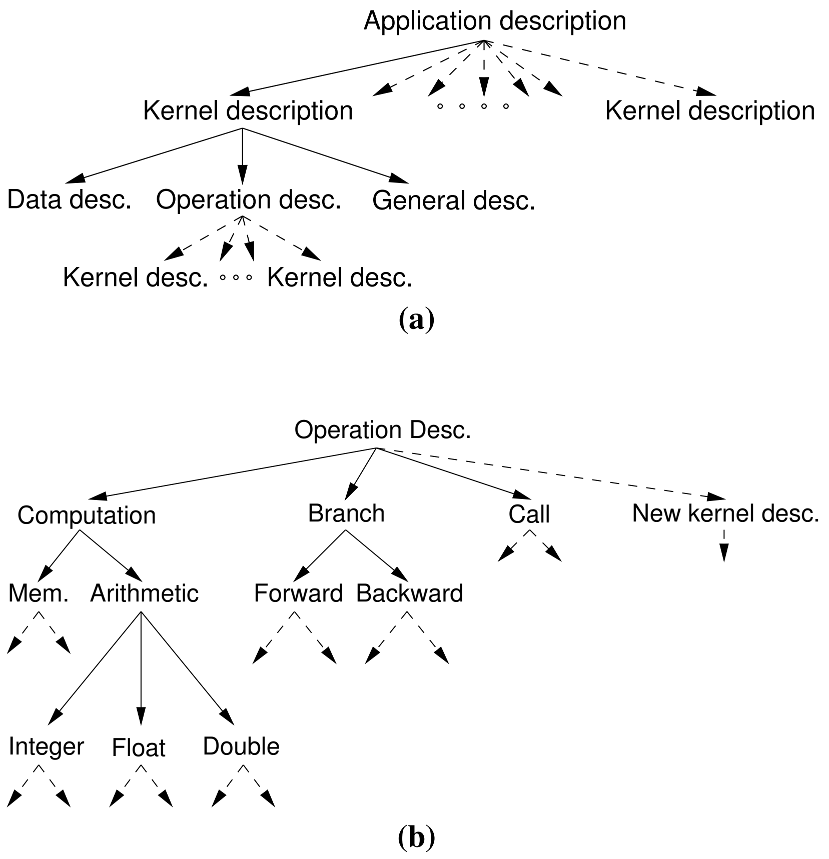


Figure 3.9: Structure of a SEA description.

from a parent kernel to be inherited by its child. Consequently, only the description fields which are really new have to be specified, thereby overwriting their old values inherited by the parent kernel.

Figure 3.9b shows (a part of) the operation description hierarchy. Each arc within the hierarchy is weighted with a certain probability. When an operation is generated, the hierarchy is traversed top-down according to the probabilities of the arcs until a leaf containing a certain operation has been reached. In order to specify the type of data on which the operation is performed (e.g. array, local variable, global variable, etc.), the data section of the kernel description is traversed in a similar manner.

Using kernel descriptions as a building block allows the modelling of an application at different abstraction levels. Basically, the behaviour of a whole program can be described with one kernel description only. In this case, the application is described at a high abstraction level, which is flexible but might not be very accurate. Alternatively, multiple kernel descriptions can be used to describe the behaviour of certain parts of a program. This approach may require somewhat more modelling effort but it represents the application behaviour with higher accuracy.

To illustrate the stochastic modelling, consider Figure 3.10. It shows a SEA description for the loop from the *ddot* code fragment of the previous section when using N=1024. For the sake of clarity, we omitted a few lines of SEA code which were not relevant to this example. The kernel, which consists of 6 instructions and starts at hexadecimal instruction address 100, is iterated 1024 times. SEA would also allow the non-probabilistic values (e.g.

```
kdescription[1024, 0x100] {
   general { instructions = 6; }
   data {
      dfp_data {
         variable[50%] {
            number = 1; global = 100%; auto = 0%;
         }
         array[50%] {
            number = 2; av_size = 1024; global = 100%;
         }
      }
      non_fp_data {
         variable[100%] {
            number = 1; global = 0%; auto = 100%;
         }
         array[0%] {}
      }
   }
   computation[100%] {
      writeback_expr[33%] {
         sfl_point = 0%;  dfl_point = 50%;
         integer   = 50%; constant  = 0%;
      }
      arith_expr[67%] {
         sfl_point = 0%;  dfl_point = 50%;
         integer   = 50%; constant  = 50%;
      }
      fp_expr {
         plain_fp[100%] {
            fadd = 50%; fmul = 50%;
            fdiv = 0%; fsub = 0%;
         }
         intrinsic_fp[0%] {}
      }
      integer_expr {
         non_bit_op[100%] {
            add = 50%; sub = 50%;
            mul = 0%; div = 0%;
         }
         bit_op[0%] {}
      }
   }
   calls[0%]         {}
   branches[0%]      {}
   sub_kdesc[0%]     {}
   communication[0%] {}
}
```

Figure 3.10: SEA description of ddot loop.

```
for (i = 0; i < N; i++) {
   a += x[i] * y[i];
}
```

Original *ddot* loop

```
1.  REG = i - N        // i < N
2.  REG = x[i] * y[i]
3.  REG = a + REG
4.  a   = REG
5.  REG = i + 1
6.  i   = REG
```

Pseudo assembly of the loop-body

Figure 3.11: Translation of the *ddot* loop into pseudo assembler instructions.

the number of instructions) to be random and behave according to some distribution.

The SEA description in Figure 3.10 has been constructed by first translating the body (including the conditional expression) of *ddot*'s loop into some sort of pseudo assembler instructions. This translation is shown in Figure 3.11. The resulting pseudo assembly consists of 6 instructions, which explains why the SEA description specifies that each kernel iteration should generate the operations for 6 instructions. The branch at the end of the loop is omitted as it is generated automatically by the STG.

The *computation* part of the SEA description basically describes the probabilities derived from these 6 pseudo assembler instructions. For example, as 2 of the 6 assembler instructions write their result back to memory, there is a 33% chance of an instruction being a writeback expression. Of these memory instructions, 50% store integer values (instruction 6), while the other 50% store doubles (instruction 4). The remaining 67% (4 out of 6 pseudo instructions) of instructions should perform arithmetics and store the results in a register (the *arith_expr* part of the description). From these instructions, 50% performs calculations on doubles (instructions 2 and 3), while the remainder operates on integers. Assuming that N is a constant, then 50% of the arithmetic instructions use a constant value as one of their operands (instructions 1 and 5). Furthermore, the *fp_expr* part of the description shows that an arithmetic instruction performing a calculation on doubles is either an addition or a multiplication (instructions 2 and 3). The remaining parts of the SEA description are constructed in an identical manner. We note that the STG typically generates multiple operations for a single pseudo assembler instruction. For example, if the STG determines that an operand of an arithmetic instruction is not located in a register, then it first generates the operations to address and fetch the operand after which it generates the arithmetic operation.

Currently, the SEA descriptions are produced by hand. This will eventually be automated by a framework in which real programs are profiled in order to generate the descriptions. Moreover, as was mentioned earlier, the stochastic trace generation for parallel platforms is not yet supported. We are still investigating how, for instance, communication can be modelled best within the SEA paradigm. A promising approach, however, is proposed by Chodnekar et al. [23]. They show that it is possible to express the communication behaviour (e.g. message generation, spatial distribution of destination nodes, etc.) in terms of commonly used distributions.

3.3 Architecture modelling

At the architecture level, Mermaid requires two different architecture models: a computational model accounting for the computational behaviour of a single node and a communication model simulating the communication and synchronisation behaviour and thus including all the nodes of the parallel platform. This approach is similar to one followed by SMART [105], which also distinguishes between a single node model and a global network model. Before describing Mermaid's architecture models, we will first give a short introduction on the object-oriented simulation language Pearl, which is used for the implementation of the models.

3.3.1 The language Pearl

The Pearl language [97] was especially designed for easily and flexibly implementing simulation models of computer architectures. It has been greatly influenced by three other languages: POOL [6], C [69] and SIMULA [12]. A Pearl program consists of a collection of *objects* running concurrently and communicating with each other via messages. This computational model of objects originates from POOL. The statement structure and the syntax of Pearl stems from C, while the object-oriented simulation model is similar to the one that is used in SIMULA. Pearl is a small language and does not support any sophisticated features, like most general purpose languages do. The idea behind this is that these features are not needed for architecture simulation.

The compilation path of a Pearl program is shown in Figure 3.12. It consists of four stages. In the first stage, the Pearl-compiler compiles the object definitions (Pearl code) to regular C code. The C-compiler then compiles the generated C code and may link ordinary C libraries with it. In the third stage, the loader interprets a defined topology of objects and loads the simulation with its given parameters on the host machine. Finally, in the last stage, the host runs the simulation, possibly using program input.

Objects

Pearl objects are processes that behave according to a specification given in a *class*; an object is said to be an instance of a class. Objects execute ordinary sequential code and have their own data space which cannot directly be modified by other objects. Instead, when an object wants to modify some remote data, it sends a message to the object with a request to

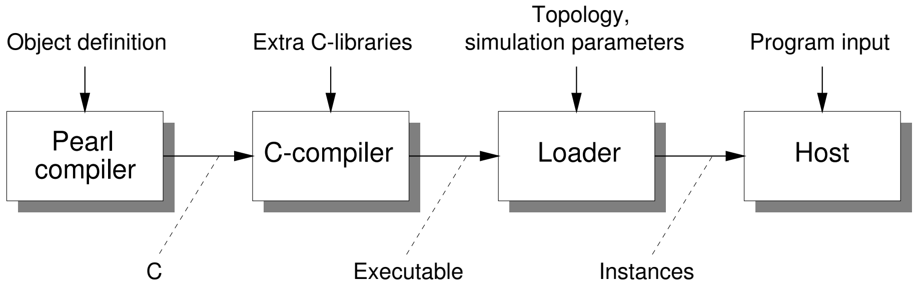


Figure 3.12: The compilation path of a Pearl program

change the data. The remote object may then change the data explicitly after receiving such a request.

Moreover, Pearl makes a clear separation between data and objects. Data is used to model the state of the hardware components (e.g. the cache state, register contents, the bus state, etc.), while the objects model the functionality of and the interactions between the hardware components. Unlike other object-oriented languages, the objects in Pearl are not dynamically created but are generated during the startup of the program. This approach originates from the fact that objects represent hardware components, which, of course, cannot be created dynamically.

Communication

Communication between objects is accomplished by either synchronous or asynchronous message-passing. Asynchronous communication continues immediately after transmission while synchronous communication waits for a reply message from the other side. An asynchronous message is composed and sent in one single action, as in the example below:

```
dest !! m_id( par_1, ... , par_n );
```

This statement places the message, which is calling method *m_id* with the parameters *par_1* ...*par_n*, in the message-queue of an object called *dest*. So, the method *m_id* should be a function in the class corresponding to the object *dest*. When the destination object decides to process the message, the message is retrieved from the queue and the function *m_id* is called inside the object with the parameters that reside in the body of the message.

The receiver of a message explicitly decides when to handle a message by executing a so-called *block* statement. This statement blocks until a message for one of the specific methods arrives. When multiple messages reside in the message queue of the object in question, the messages are handled in a FIFO order. For example

```
block( method1, method2 );
```

blocks until a message for either method *method1* or *method2* arrives. When the message arrives, the appropriate method is called after which control is returned to the statement after the *block* statement. The keyword *any* may be used to indicate all possible methods, while a call to *block* without parameters will block forever.

The synchronous send is denoted using a single exclamation mark:

```
x = dest ! m_id( par_1, ... , par_n );
```

A reply is sent back within the receiving method by using the statement:

```
reply( a_value );
```

Blocking for synchronous messages is identical to the blocking scheme discussed for asynchronous communication.

Virtual time

Because of its simulation nature, Pearl is equipped with a *virtual clock*. During the run of a Pearl program, this clock always holds to the current simulation time. This clock neither can go backwards in time nor does it flow continuously, which implies that Pearl obeys a discrete-event model. An object can indicate that it wants to wait an interval in simulated time and does not want to be re-scheduled during that period. For example, an object that wants to wait 5 time units executes:

```
   blockt( 5 );
```

When all objects are waiting for the virtual clock or messages, the clock is advanced to the first time in the future where some object will become active. If all objects are waiting for messages, implying that there is a deadlock situation and none of the objects will be active again in the future, then the simulation is said to be terminated.

3.3.2 Single-node computational model

We used Pearl to implement Mermaid's two *template architecture models*: a single-node computational model and a multi-node communication model. The first of these models, the single-node computational model, simulates the processor(s) and the memory hierarchy of a multicomputer node. It is generic in the sense that a wide range of node architectures can be represented by means of parameterisation. Figure 3.13a depicts the computational template model in which roughly each component (i.e. box) is implemented by a Pearl object.

The CPU component simulates a microprocessor within the node architecture. It supports the operation set described in section 3.1.3. Each operation is associated with a fixed execution latency. The CPU is parameterised with the operation latencies in clock cycles and the clock speed. Furthermore, the CPU component can be configured to include more advanced features such as a model for superscalar processing. The inclusion of these details generally results in improved accuracy and, inherent to that, a more computationally intensive simulation. Nevertheless, a cycle-accurate simulation of instruction pipelines is not feasible due to the lack of information in the operations.

The cache hierarchy component simulates the first-level cache and, if available, also the higher-level caches of the memory hierarchy. It is parameterised with the cache dimensions, associativity, block replacement strategy, write strategy and the latencies associated with

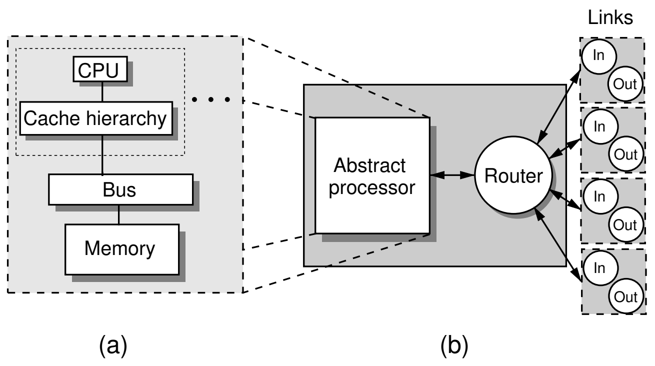


Figure 3.13: The template architecture models.

cache hits, misses and replacements. Moreover, a cache can either be configured to work as an instruction/data cache or as a unified cache.

The single-node model supports a setup of multiple processors using a common cache hierarchy. To guarantee cache consistency in such a configuration, the caches provide a coherency protocol [133]. By default, this is a snoopy cache protocol. But other strategies, like directory schemes, can be added with relative ease.

To connect the processors and the cache hierarchy to the memory, the template model defines a bus component. It is a simple forwarding mechanism, carrying out arbitration upon multiple accesses. The parameters used to configure this component include the bus-width, bus cycle-time and arbitration details. Changing the bus to a more complex structure, such as a multistage network, can be done without much remodelling effort. In that case, only a new Pearl module needs to be written, replacing the bus component within the template model.

Finally, the memory component simulates a simple DRAM memory. It is parameterised with memory size, memory refresh rate, and memory access latencies.

3.3.3 Multi-node communication model

A node within the communication template model is constructed from an abstract processor, a router and multiple communication links. This setup is shown in Figure 3.13b. The nodes are connected in a topology that reflects the physical interconnection scheme of the multicomputer, resulting in a multi-node simulation model.

Each abstract processor within the multi-node model reads an incoming operation-trace, processes the *compute* operations and dispatches the communication requests to a router component. After this point, the router is responsible for further handling the transmission. This may include splitting up messages into multiple packets. Furthermore, the router component routes the resulting and all other incoming messages (packets) through the communication network. For this purpose, it uses a configurable routing and switching strategy. Currently, only routing algorithms for two-dimensional mesh networks are available: deterministic routing based on dimension ordering and partially adaptive negative-first routing [100, 101]. Other routing schemes can be added and evaluated by simply plugging in a new router component.

Parameters also include communication link bandwidth, packet size, routing hop latency, message setup and processing delays and buffering specifics. For packet switching, the routers support both the store-and-forward [100] and the wormhole routing [31] techniques. A more elaborate discussion of these switching techniques is presented in Chapter 5.

3.3.4 Putting it all together

Detailed simulation of a multicomputer architecture requires that the single-node computational model is replicated for each of the nodes taking part in the simulation. Each instance of the single-node model is then assigned to a node within the communication model in order to feed it with the computational tasks and communication operations. This is illustrated in Figure 3.14, which essentially is a more detailed version of Figure 3.4.

The multi-node communication model, with its message passing, intrinsically suggests that the system under investigation should belong to the class of multicomputer architec-

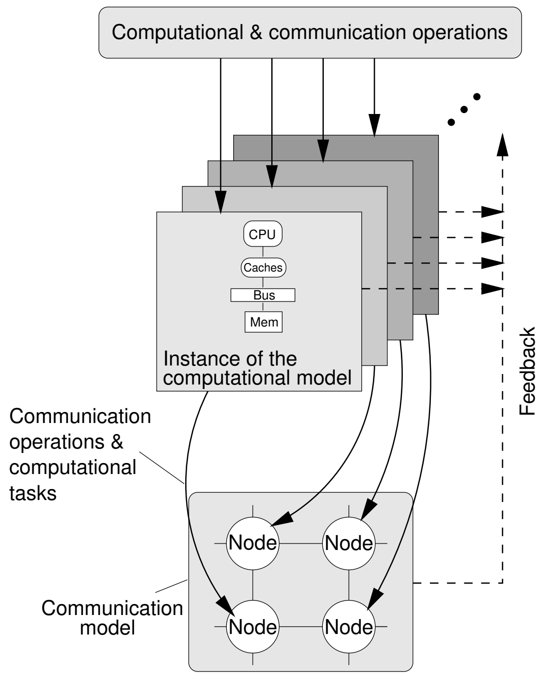

Computational & communication operations
CPU
Caches
Bus
Mem
Instance of the computational model
Feedback
Communication operations & computational tasks
Node
Node
Node
Node
Communication model

Figure 3.14: Detailed simulation of a distributed memory multicomputer in Mermaid.

tures. But, by only using the computational model and configuring it with multiple processors, a symmetric multiprocessor (SMP) can be simulated. Note, however, that this would require the addition of atomic operations to the current set of computational operations. A disadvantage of this approach is that simulation can only be performed at the level of computational operations, being the highest level of detail.

Hybrid architectures, such as the ones belonging to the increasingly popular class of CCNUMA machines [80, 86], can be modelled by both defining multiple processors on a node sharing a memory and using the communication model to interconnect multiple of these SMP clusters in a message-passing network.

Mermaid's simulation methodology is in many aspects similar to that of the SMART simulation environment [105], which was developed at roughly the same time as Mermaid. There are, however, several important differences between both simulation environments. For example, SMART initially focused on the simulation of communication and its consequent computation. Simulation of the application's computation (i.e. the local instructions) could not or could only partially be performed. Only recently, extensions have been made which make it possible to fully simulate parallel platforms [104]. This is in contrast to Mermaid, which has always supported the simulation of the whole parallel platform. Moreover, there is a difference with respect to the generation of the simulation events. In SMART, the target assembler code is augmented, while Mermaid uses the higher, and thus less architecture dependent, C source level for augmentation. Besides this, Mermaid does not perform augmentation at all in the case it uses the stochastic application model.

3.4 Analysis of simulation output[†]

Computer architecture simulations typically produce a large quantity of simulation output offering statistics on all parts of the modelled architecture. These statistics require post-mortem analysis, which often is a tedious task because of the bulkiness of the data. For this reason, we have built a tool which allows for extracting the required data from the simulation output and which, in addition, can perform all kinds of statistical methods on the obtained data. The tool, which is called RAPID (RAPid Interpretation of Data), contains a specification language in which the user specifies what statistical methods should be used on which parts of the simulation data. According to this specification, RAPID generates an executable performing the specified types of analysis. A comprehensive description of RAPID can be found in [110].

Post-mortem analysis may, however, not be sufficient to gain a good insight into the performance behaviour of a computer architecture. In some situations, it might be helpful, or even essential, to capture runtime effects which have an important influence on the overall performance. Studying these runtime effects may result in a better understanding of *why* the performance of the simulated architecture is what it is. Therefore, Mermaid also offers a Graphical User Interface (GUI) supporting the runtime visualisation of architecture simulations.

3.4.1 The GUI–support

At the design-phase of the GUI, we identified three important requirements to which the GUI should adhere: flexibility, clarity and applicability. Regarding flexibility, the GUI must not be geared towards one specific type of architectural model or simulation. As Mermaid's architectural models can easily be adapted, the GUI should allow the visualisation of basically any computer architecture. For this reason, we decided to offer the GUI-functionality at the Pearl level rather than at a higher level, such as at the level of Mermaid's architectural template models. Moreover, the GUI must be user friendly. Therefore, it should be able to display the required data in a clear manner without burying the user under loads of statistical information. Finally, the GUI-support must be easily applicable and, of course, it should be equally simple to leave out the visualisation without being hit by a performance penalty. To achieve this, the GUI is defined *entirely independent* of the Pearl code. A special GUI-description, written in *Agile* (A Graphical user Interface LanguagE), specifies the structure and the functionality of the GUI. It describes, for example, which data values should be extracted from the simulation and how they should be visualised. The Pearl runtime system uses this description to take care of the actual visualisation. So, in this approach, the Pearl code is not polluted by all kinds of graphics-primitives. As a consequence, we can use the GUI-support for older simulations without the need to modify the Pearl code. A more detailed overview of Agile is beyond the scope of this thesis. The interested reader is therefore referred to [70].

[†]This work is based on [71].

	Discrete	Continuous
Temporal behaviour	**snapshot values** — values that say something about the simulation at a particular moment	**integrated values** — values that say something about the simulation over some time interval
Spatial behaviour	**event values** — values that describe an event being raised or not	**quantitative values** — values with a large domain

Table 3.3: Classification of values according to their temporal and spatial behaviour.

The basic framework of the GUI, as obtained by the Agile description, consists of a so-called canvas (the main window) with a few controls which make it possible to run the simulation in three different modes: continuously, per time step and per scheduled event. On the canvas, small windows can be placed representing the different components of the computer architecture. In this scheme, every Pearl object is attached to one window. However, if an object is not interesting, its window can be omitted. Visualising widgets, or in short *visuals*, can be placed inside the object-windows. These visuals, which come in various types, are the building blocks for displaying statistics from the objects. Using this framework, the construction of a GUI is composed of three steps:

1. Determining which data values from the simulation are interesting.
2. Doing (optional) transformations on these values.
3. Determining how to present the (possibly transformed) values. In other words, specifying which visuals should be used.

Retrieving and transforming data values

Values which are retrieved from a running simulation can, of course, be directly displayed. This is, however, not very sensible as most values first require a transformation of some sort before they actually represent meaningful information. For example, most of the values that are retrieved from the simulation have a temporal character; they only provide information on the simulation at the moment that they were derived and therefore tend to change very fast. When going through a simulation step by step to get a better understanding of what is happening, such a value can be directly displayed. But when the simulation is run continuously, its direct visualisation is not desirable. Instead, one would like to see the value's evolution over some time interval. So, we should realise that there are different types of data values, each having its own characteristics and thus requiring different visualisation methods. In Table 3.3, we present a classification of data values according to their *spatial* and *temporal* characteristics: a value can either be *discrete* or *continuous* in both time and space. Discrete values have a small domain (discrete in space) or the length of the simulation time frame on which these values hold information is limited (discrete in time). On the other hand, continuous values have a large domain (continuous in space) or hold information for an arbitrary period of simulation time (continuous in time).

Many of the data values are snapshot values, which cannot be directly visualised when simulating in continuous mode. These snapshot values should first be translated into integrated values. For this purpose, special *transformer functions* are used, which sample a

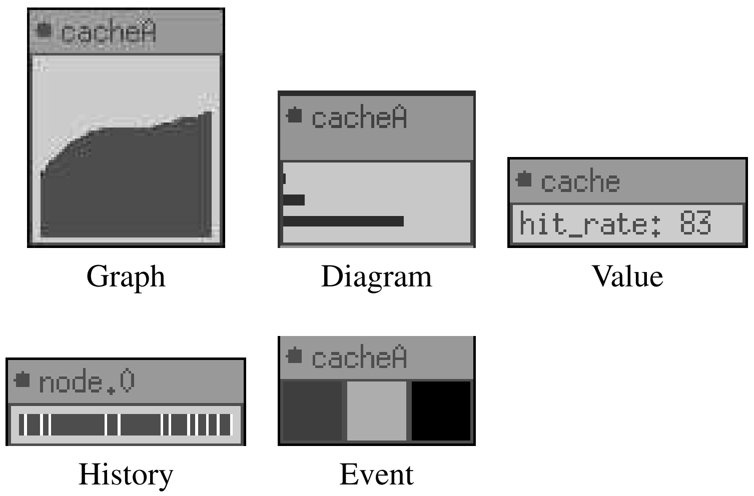


Figure 3.15: The visual-types.

snapshot value at regular intervals and, as a consequence, determine the behaviour of the value over a longer period of time. Currently, there are five transformer functions: *smooth*, *history*, *sum*, *average* and *group*. Applying the *smooth* transformer, a value is smoothed by sampling it at regular intervals and calculating its new value from its previous sample and its current value according to a certain weight. The *history* transformer also samples a value on regular intervals but, unlike *smooth*, stores these samples. This history of samples can then be used for further processing with transformers or it can be passed directly to the GUI. In case of the latter, the entire range will be displayed in one visual. The *sum* and *average* transformers are strongly related to the *history* transformer. In fact, the *sum* and *average* are only defined on *histories*. Whereas *history* maps one value to a range of them, *sum* and *average* map a range to one value. The names are self-explanatory: *sum* sums up all the values in a range, while *average* calculates the arithmetic mean. Finally, values can be grouped together to display them in one visual by means of the *group* transformer.

Visuals

The visuals which are capable of displaying the (transformed) values are shown in Figure 3.15. Each of these visuals uses one of three different ways to display a value: by a colour, by the length of a bar or verbatim. Using the length of a bar is convenient when displaying quantitative values. On the other hand, event values can often be interpreted more easily when they are represented by a colour rather than the length of a bar. In the case a simulation is executed in step-mode, an event can be displayed as being raised or not in two colours. When simulating in continuous mode, the amount of time an event was raised can be displayed as a range of colours, but also as the length of a bar. The choice between those two depends on the characteristics of the value and what the user wants to detect from them.

The two visuals that use the length of a bar to display a value are called *graph* and *diagram*. The *diagram* visual takes a group of values and draws a number of bars of which the lengths are relative to the values. As the value changes, the length of the bar changes as

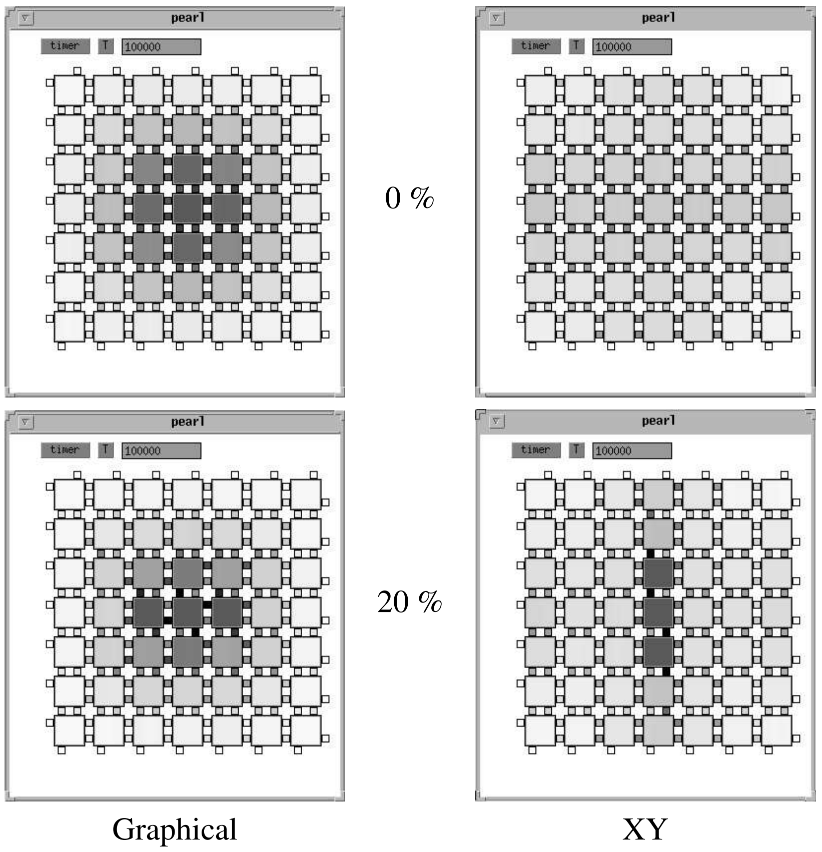


Figure 3.16: Visualising XY and Graphical routing in a mesh network.

well. The *graph* shows the recent past of a value. It scrolls from right to left and new values are being drawn on the right. The metrics of the value are omitted because they may easily reduce the clarity of the GUI.

The visuals *history* and *event* use a colour to display a value. When simulating step-by-step, these visuals typically display event-like values: *event* is used to display events as being raised or not and *history* displays their recent past. However, the *history* and *event* visuals can do more. When simulating in continuous mode, it might also be interesting to use them to display quantitative values. This can be done by assigning a colour to the upper and lower bound of the value and have a number of shades between the two extremes. With this technique, a large number of values from different objects can be compared in an instance. Figure 3.16 gives an example of this type of visualisation. It shows the congestion in a mesh network by colouring the routers and the channels within the network. Dark grey values denote high congestion, whereas the light grey values indicate that there is (almost) no congestion. In reality, these are real colours ranging from from blue (no congestion) to red (high congestion). The two pictures on the right show the network behaviour when packets are routed using an XY routing scheme [100, 101], while the pictures on the left show the behaviour using Graphical routing [7]. XY routing is a technique based on dimension ordering (first route a packet in the X coordinate, then in the Y coordinate), whereas Graphical routing is based on Bresenham's algorithm to draw lines as known from the field of computer graphics [40]. The pictures at the top are for a uniform communication load, whereas the pictures at the bottom show the behaviour for a communication load in which a hotspot in the center of the mesh is introduced. This hotspot consumes 20% of all message traffic. It is not our intension to explain the pictures in detail, but the point we want to make is that the pictures clearly suggest that XY routing outperforms Graphical routing for a mesh network and for the applied communication loads. Thus, using visualisation, this result is noticeable instantly, while this may not be true when a vast amount of simulation data has to be analysed post-mortem. The interested reader is referred to [71] for a more detailed discussion of this example.

The last visual type, called *value*, directly displays a value. A frequent use of this visual will result in a decrease of the clarity of the GUI. It should therefore be used with some care.

3.5 Summary

In this chapter, we presented the Mermaid framework for the performance evaluation of MIMD multicomputer architectures. The simulation environment allows for study of the interaction between software and hardware at different levels, ranging from the application level to the runtime system level. Architecture simulation is supported at various abstraction levels. For example, if only fast prototyping is required, then simulation can be performed at a high level of abstraction. However, if accuracy is required then the simulation environment is capable of simulating at a lower, but less efficient, level of abstraction.

Mermaid strives to support the evaluation of a wide range of architectural design options. To allow this modelling freedom while delivering good simulation performance, detailed simulation is performed at the level of abstract machine instructions rather than at the level of real instructions. For this purpose, we use an execution-driven simulation technique that is strongly related to traditional trace-driven simulation. The traces driving the simulator consist of events, called *operations*, which represent processor activity, memory I/O or message-passing communication. To guarantee the validity of these multiprocessor traces, the trace generation can be controlled by the simulation of the target architecture. In other words, the trace generator is interleaved with the architecture simulator.

Because Mermaid's architectural simulators are operation-driven and therefore do not interpret real machine instructions, application behaviour must be modelled explicitly. This workload modelling can be done in two ways. In the first approach, real applications are instrumented in order to trace their execution behaviour. The instrumentation of the programs takes place directly at the C source level to reduce the architecture dependence of the application model. This basically allows us to use the same augmented application for the evaluation of any multicomputer architecture. In the second approach, application behaviour is modelled using probabilistic descriptions. Such a scheme provides more flexibility than the tracing of real programs but its accuracy is only modest. When fast-prototyping new architectures, however, this may be a useful technique.

To facilitate the analysis of the simulation output, Mermaid features tools for both the post-mortem and run-time analysis of data. In the case of the latter, the user is allowed to define a GUI visualising the run-time effects which may have impact on the performance of the computer architecture.

Chapter 4

On the accuracy and efficiency of Mermaid[†]

"Even a stopped clock is right twice a day."
Source Unknown

Modelling in general, and so the modelling of computer architectures, always requires the model to be validated. To put this in the context of the above quote, we should find out whether or not the model is "more than a stopped clock". Without any knowledge on the accuracy of the simulation, the computer architect cannot perform design space exploration in a sensible manner. An inaccurate model may lead to wrong design decisions, harming the architecture's performance or cost, or both. In this chapter, we will therefore address the validation of our simulation methodology. This validation study purely focuses on reality-based workloads as Mermaid's stochastic modelling of workloads is still subject to many (fundamental) changes. The validation of our stochastic workload modelling framework is future work.

In the previous chapter, we also recognised that good simulation efficiency is essential for an architectural simulator. To evaluate whether or not we have succeeded in building an efficient simulation environment, this chapter presents an additional study on Mermaid's performance. Furthermore, we suggest several methods to improve the simulation performance. Of these methods, we have implemented one: Parallel Mermaid, which allows distributed simulation on a pool of workstations.

4.1 Validation

To gain insight in the capabilities of our simulation methodology with respect to accuracy, we have modelled an existing distributed memory multicomputer. This machine, a Parsytec GCel, consists of Inmos T805 transputers connected in a 2D grid network. The following section briefly describes the architecture model that has been built for the Parsytec multicomputer and presents a set of validation experiments.

[†]This chapter is based on [108, 111].

Channel throughput	1.7 Mbyte/s	Routing overhead	2 μs
Receive overhead	25 μs	Inter-board penalty per byte	0.25 μs
Send overhead	47.5 μs	Context switch	0.75 μs
Packet creation overhead	10 μs	Memory latency per word	180 ns

Table 4.1: Parameters of the Parsytec GCel communication model.

4.1.1 The architecture model

For the architecture model of the Parsytec GCel multicomputer, we used Mermaid's hybrid machine model as shown in Figure 3.14. Each instance of the computational model simulates a T805 transputer. The configuration of the computational model is straightforward as it only consists of three components: a processor core, a bus and a DRAM. There are no caches[1] that need to be modelled. The latencies of the computational operations were derived from both measurement on a real transputer and from technical documentation.

Regarding the communication model, we configured it to represent a mesh topology performing store-and-forward XY routing. On the real machine, this routing is entirely done in software. This implies that whenever the transputer routes an incoming packet, a context switch is performed to a system process routing the packet after which the transputer switches back to the user process. Context switches can, however, be performed relatively fast on transputers as they have some kind of hardware mechanism for context switching.

The entire set of parameters for the communication model is listed in Table 4.1. A difficulty in determining the values for these parameters is the fact that the Parsytec's operating system has been a black box to us. So, the OS overheads of message-passing communication (e.g. communication setup overheads) were not known. We therefore tried to determine these overheads by means of experimentation. All parameters in Table 4.1 should be self-explanatory, except for the inter-board penalty. This penalty is caused by the physical network organisation of the Parsytec GCel multicomputer. The mesh network is constructed by boards containing clusters of 16 processors. We found that inter-board communication adds a significant latency in comparison with intra-board communication.

4.1.2 Experiments

The validation has been performed by comparing the simulation results of several benchmarks with the results of real execution. Initially, the computational and communication models were validated separately. Table 4.2 shows the overall results of these experiments using *normalised errors*. The normalised error, referred to as $\widehat{error}$, is the ratio between the actual and the predicted execution time and is defined as

$$\widehat{error} = \frac{\max(T_{real}, T_{simulation})}{\min(T_{real}, T_{simulation})}$$

[1]The T805 transputer contains a small addressable SRAM which can be used as some sort of cache. This memory is not included in our model.

where T_{real} is the actual time measured on the real system and $T_{simulation}$ is the predicted time. So, an $\widehat{error}$ of 1 implies that there is no modelling error. Table 4.2 gives the average $\widehat{error}$ for several experiments (calculated using the geometric mean), the standard deviation σ of this average error and the worst case $\widehat{error}$. The workloads for the computational model consisted of a set of well-known numerical kernel functions, including *ddot* (double precision innerproduct), *daxpy* (double precision vector update) and some of the Lawrence Livermore kernels [89].

The communication workloads, which only model message passing and its consequent (computational) delays, can be divided into three categories. The first category, namely that of light communication loads, uses benchmarks performing message roundtrips. The two remaining types of communication workloads are for so-called "stress-testing" purposes, as they heavily congest the network. These two workloads can again be subdivided into uniform and non-uniform loads. The first type of load uniformly distributes communication over the network, while the latter type only stresses some small regions within the network, causing hot-spots to appear.

For the computation and light communication loads, the average $\widehat{error}$ is small as it does not exceed 1.015. More importantly, the standard deviation and the worst case error indicate that the performance estimates for these loads are quite accurate in general. The errors of the stress-testing communication benchmarks are higher, especially in the case of the non-uniform loads. This can be explained by the fact that these benchmarks produce workloads exhibiting high contention for network resources, which tends to amplify any existing modelling errors. While a worst case $\widehat{error}$ of nearly 1.30 was measured for these extreme loads, the standard deviations suggest that the accuracy in the general case is tolerable.

Validation of the integral system, consisting of both the computational and communication models, has been performed by three SPMD-type benchmarks which are listed in Table 4.3: *gauss* (a solver of linear equations), *pdmm* (a matrix multiplication) and *sort* (an integer sort). Of these three benchmarks, none requires execution-driven simulation (i.e. they all contain fixed communication patterns). This implies that, in all three cases, the simulation can be performed in pure trace-driven mode. Furthermore, only the execution of *sort* is data dependent. Consequently, when simulating *sort*, the trace generator threads need to synchronise at the application level (see Section 3.1.2).

Average $\widehat{error}$	σ	Worst-case $\widehat{error}$	Workload
1.015	0.013	1.038	Computational loads
1.015	0.013	1.050	Light communication loads
1.042	0.031	1.109	Uniform congesting communication loads
1.088	0.092	1.297	Non-uniform congesting communication loads

Table 4.2: Validation of the separate computational and communication models.

Benchmark	Description	Data input
Gauss	A solver of linear equations using Gaussian elimination	$n \times m$ matrices
Pdmm	A double-precision matrix multiplication	$n \times m$ matrices
Sort	An integer odd-even transposition sort	nK of integers

Table 4.3: The parallel benchmark applications used for validation.

Figure 4.1 depicts the execution times and estimated (simulation) times of the benchmarks for a range of different data sizes. With a calculated average $\widehat{error}$ of 1.022 and a standard deviation of 0.029 for *gauss*, high accuracy is obtained. For the simulation of *pdmm*, we measured a somewhat larger average $\widehat{error}$ of 1.040 and a standard deviation of 0.035. Finally, in the case of *sort*, we achieved an average $\widehat{error}$ of only 1.019 with a σ of 0.019. So, again, good accuracy is obtained. The worst case $\widehat{error}$ for *pdmm*, *gauss* and *sort* equal to 1.181, 1.115 and 1.077 respectively.

Another important observation is, of course, that the simulation closely follows the execution trend. This is even the case for the small irregularity in the graph of *gauss* at a problem size of 100×100 using 32 processors. Closer examination showed that this irregularity is caused by a load imbalance.

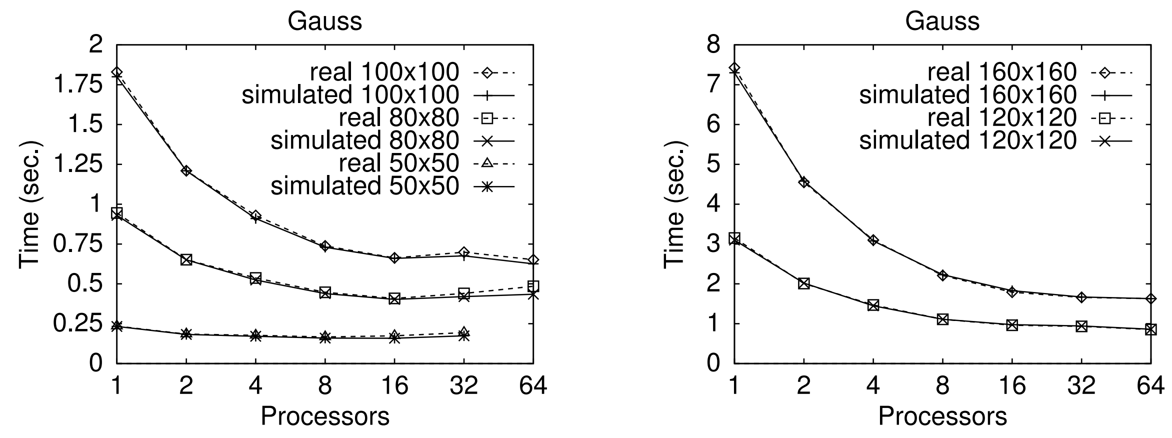
4.1.3 Discussion

The validation experiments demonstrate that, despite of the high abstraction level, good accuracy can be obtained. Nearly all measured values of $\widehat{error}$ (except for the non-uniform stress-testing workload) are below 1.05, which means that we more or less achieved our accuracy-goal of keeping the average errors below the 5% (see Chapter 3). Naturally, this conclusion is only appropriate for the model we investigated. The process of validation must therefore be a continuous effort which is repeated every time a model is constructed or changed.

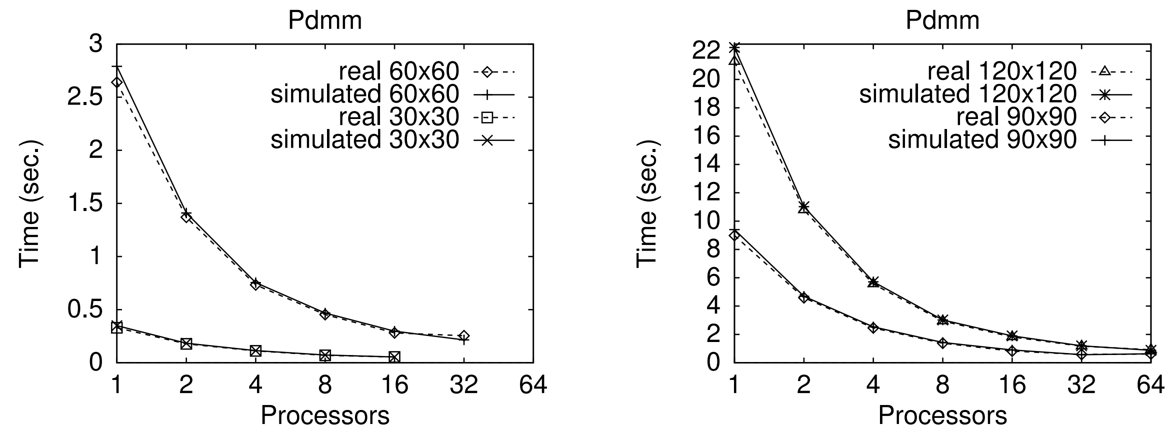
In the Mermaid framework, three areas can be distinguished where validation is necessary:

- Is the translation from annotations to operations accurate for a given target processor, runtime model and compiler?
- Does stochastic generation of a trace of operations resemble the behaviour of real applications?
- Does the architecture model resemble the latencies of the real operations under different loads of operations?

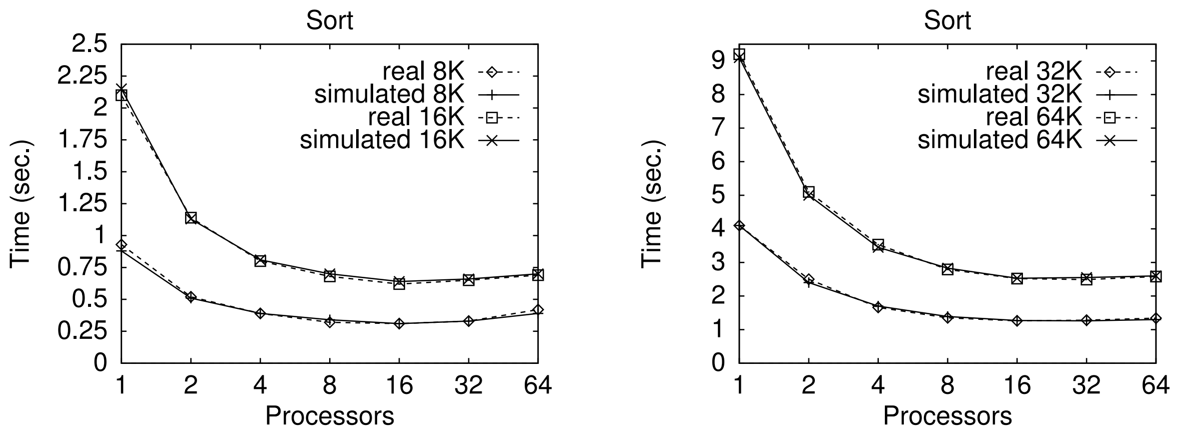
Validating the translation of annotations to operations can be done by a comparison of generated assembly code and generated operations, if a compiler for the target processor is



(a)



(b)



(c)

Figure 4.1: Validation results of *gauss* (a), *pdmm* (b) and *sort* (c).

available. Specific compiler optimisations can be detected, which may result in tuning the annotations or even in adjusting the trace generator.

To verify the accuracy of the stochastic generation of operations, the performance critical parts of a typical application can be instrumented to generate a reality-based operation-trace of the application. From this trace, the amount and locality of the application's data and instruction accesses, its arithmetic behaviour and its communication behaviour can be quantified. The resulting quantification can then be used during or after the stochastic generation as a semantic accuracy check. Evidently, this is not a trivial task since, for instance, locality of reference is hard to quantify. There is still no clear metric for locality, although some metrics have been proposed [19]. Hence, this topic requires additional research.

On the other hand, the stochastic trace generator does not always have to operate strictly within the range of application behaviour, as there is the additional interest in studying the performance degradation when applying extreme workloads.

Finally, to validate the architecture models, one typically simulates a suite of reality-based benchmarks and compares the results with real execution. These benchmarks can be simple and only concentrate on rudimentary architecture operations. Once these operations have been correctly calibrated, the validation can be extended to cover larger and more complicated benchmarks or applications.

Naturally, this type of validation of the architecture models requires (parts of) the target architecture to be available. Unfortunately, this is often either not the case or the available parts are hard to study in isolation. As a result, the validation is mainly restricted to the visual inspection of the model code by the programmer. Additionally, in some cases, simple benchmarks of which the execution behaviour is well-understood might be used to act as a sanity check [13].

4.2 Simulation performance

In order to indicate the performance of computer architecture simulations, it is common use to calculate the *slowdown* of the simulator. This metric is an indication of the overhead introduced by the simulator as opposed to real execution on the target platform. An exact value for the slowdown of a simulator cannot be given since it may depend on the host machine and the type of application and architecture that are being simulated. Therefore, a *typical* value is generally used.

There are several ways to calculate the slowdown. The simplest one, and the most commonly used, is by calculating the fraction between the simulation time needed by the host computer and the time the target computer would have needed:

$$slowdown_{time} = \frac{\text{execution time}_{host}}{\text{execution time}_{target}} \tag{4.1}$$

The problem with this slowdown calculation is that it probably says as much about the host's performance as about the efficiency of the simulator. More specifically, the calculation is entirely based on execution time and therefore disregards the differences between the host and target architectures.

To reduce the slowdown's dependency on the performance of the host platform, some researchers apply a somewhat different calculation which is based on *cycles* rather than on time. In this case, the slowdown is defined by the number of cycles it takes for the host computer to simulate one cycle of the target architecture:

$$slowdown_{cycle} = \frac{\text{execution time}_{host}}{\text{cycle time}_{host}} \times \frac{\text{cycle time}_{target}}{\text{execution time}_{target}} \tag{4.2}$$

As this calculation takes the cycle time of both the host and target architectures into account, it is more meaningful than Equation 4.1. However, it is still architecture dependent as the host and target architectures may perform different amounts of work in one cycle. So, ideally, the slowdown calculation would specify the simulation overhead in the number of instructions required by the simulator to simulate one instruction on the target architecture. This means that the number of Cycles Per Instruction (CPI) of both the host and target platforms should also be taken into account:

$$slowdown_{instr} = \frac{\text{execution time}_{host}}{\text{cycle time}_{host} \times \text{CPI}_{host}} \times \frac{\text{cycle time}_{target} \times \text{CPI}_{target}}{\text{execution time}_{target}} \tag{4.3}$$

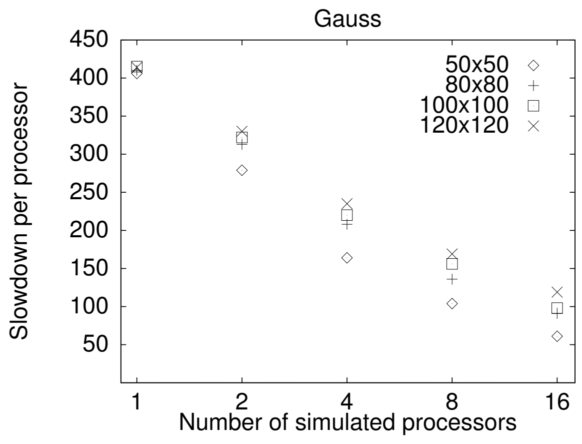
But, because average CPI values often are ambiguous and hard to obtain, the $slowdown_{instr}$ metric is hardly ever used. Therefore, we use $slowdown_{cycle}$ as the metric for our simulation performance throughout the rest of this chapter.

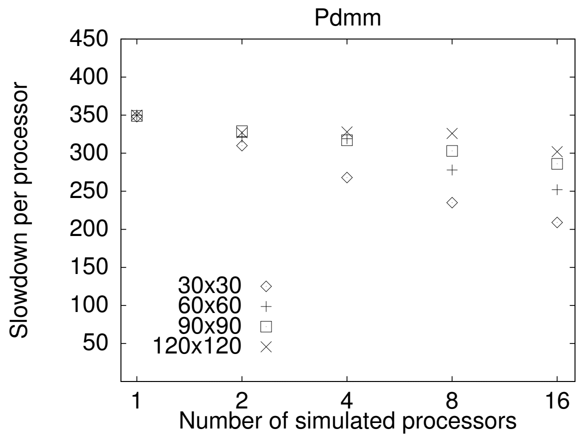
4.2.1 Mermaid's performance

To determine the simulation performance of Mermaid, we examined both the multicomputer simulator of the previous section and a simulation model of a Motorola PowerPC 604 using two levels of cache. For a mix of application loads, we measured a typical $slowdown_{cycle}$ of 60 to 650 per simulated processor. So, an Ultra Sparc processor running at 143Mhz roughly simulates between 220,000 and 2,400,000 cycles per second. The slowdown of the simulation of an entire multicomputer is a function of the slowdown of a single processor and the overhead caused by simulating the interconnection network. Typically, this slowdown is calculated by multiplying the per-processor slowdown with the number of simulated processors. Thus, a simulated multicomputer containing 64 nodes would then have a slowdown of somewhere between 4,000 and 40,000.

Figure 4.2 shows the $slowdown_{cycle}$ per processor as a function of the number of simulated processors for the three benchmarks of the previous section. All graphs more or less suggest that the slowdown decreases when simulating more processors. This is caused by the fact that simulation of communication is quite efficient in Mermaid, whereas computation is simulated more low-level and therefore more expensive. In these particular experiments, the simulation of communication is even faster than the communication within the real machine. Therefore, the slowdown decreases as the number of nodes increases.

To have a reference for comparison, Table 4.4 shows the typical slowdowns per processor of several of the multiprocessor/multicomputer simulators which were already discussed in Section 2.3.1. We should note, however, that the comparison must be performed with care. This because the effect of different compilers, host platforms, operating systems and target architectures which cannot be isolated and therefore devaluates a direct comparison between the various simulators. Table 4.4 is split into two parts; one part contains the simulators for which the slowdown was calculated based on cycles while the second part contains slowdowns of simulators based on time.





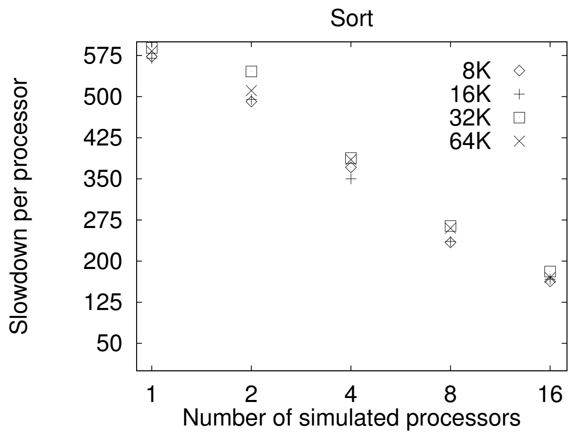


Figure 4.2: Slowdown per processor versus the number of simulated processors.

To recapitulate, except for Talisman, all simulators listed in Table 4.4 apply direct execution. Talisman is an extremely fast instruction-level simulator which is highly geared towards the simulation of one architecture: the Meerkat multicomputer [8]. Both the Wisconsin Wind Tunnel (WWT) and Tango Lite use, besides direct execution, distributed simulation to boost performance (Tango Lite is the distributed implementation of Tango [33]). For SimOS, it is quite hard to determine the typical slowdown. Like Mermaid, it also provides several levels at which can be simulated. SimOS' less detailed level of simulation ob-

Simulator	Typical slowdown per processor
Cycle-based slowdown	
WWT [117, 96]	2–60
FAST [16]	10–100
Talisman [9]	100–150
Mermaid	60–650
Time-based slowdown	
Tango Lite [45]	15–40
SimOS [119, 118]	2–250
Proteus [18]	35–100
SPAM [63]	15–425
Mermaid	13–600
Tango [33]	500–2000

Table 4.4: Typical slowdowns of several multicomputer/multiprocessor simulators.

tains only very small slowdowns, whereas its most detailed mode of simulation may reach slowdowns of up to 13,000 per processor. We did not consider this detailed mode in Table 4.4.

The numbers from Table 4.4 suggest that Mermaid is slightly slower than most of the other simulators. Apparently, we have traded some some performance for the extra flexibility obtained by operation-driven simulation. However, the performance numbers we have presented until now are those for its most detailed mode of simulation. If fast prototyping of a multicomputer is the primary goal, then Mermaid's communication model can be used directly. The slowdown of this type of simulation depends on the amount of computation and communication present within the application. Computation can be simulated extremely fast since it is modelled at the level of tasks, whereas communication is simulated in more detail and is thus less efficient. Our measurements indicate that simulation at this level of abstraction results in a typical $slowdown_{cycle}$ of between 0.5 and 4 per processor. This means that an entire multicomputer can be simulated with only a minor slowdown and, in some cases, even with a speedup! Comparing these numbers to the ones listed in Table 4.4, one can conclude that using this mode of simulation, Mermaid is significantly faster than any of the other simulators.

We believe that the simulation efficiency can still be enhanced, making Mermaid even more competitive performance-wise. For instance, the Pearl simulation language, in which the architecture models are written, emphasises the modularity and easy implementation of architecture models. It generates only moderately efficient code. The choice of a general-purpose language to implement our models might therefore improve the simulation performance. However, this would probably harm Mermaid's flexibility which is, in our opinion, undesirable.

A more promising approach is to exploit the inherent parallelism found in simulations of parallel computers. By performing *distributed* simulation, performance might be gained without sacrificing any accuracy. Results from the WWT and Tango Lite have shown that distributed simulation may improve the simulation performance considerably [117, 96, 45]. In fact, the WWT is presently one of the fastest, if not the fastest, multiprocessor simulator available. This performance gain can subsequently be used for the study of larger target architecture configurations and more realistic applications. As a side-effect, distributed simulation also improves the simulator's scalability with respect to the memory consumption. Because the memory requirements are spread over multiple host computers, it is possible to simulate workloads that consume large amounts of memory, which is common for multicomputer applications.

We believe that the next logical step in improving Mermaid's efficiency and scalability is to extend it in order to support the distributed simulation of multicomputer architectures. For this reason, we implemented a prototype of *parallel Mermaid* which allows the simulation to be distributed and executed on a cluster of workstations. In the next section, we present an overview of this parallel Mermaid version, discussing its design issues and its performance.

4.3 Parallel Mermaid

Mermaid's hybrid architecture model, as depicted in Figure 3.4, exhibits a lot of inherent parallelism, which simplifies the parallelisation. The instances of the computational model perform computations which are local to a single node only. So, these instances are independent of each other (i.e. their synchronisation and communication is simulated in the communication model only). As a consequence, they can easily be simulated in parallel on different hosts. Normally, distributed simulation requires extra measures to guarantee the causality within the simulated system [75]. However, in our simulation methodology, all events which are global to the distributed nodes are simulated by the centralised communication model. So, when the communication model is not parallelised and is executed on a single host, it can correctly sequentialise the incoming communication requests, thereby keeping one global notion of simulation time. In other words, the sequential communication model synchronises the distributed simulation clocks of the computational model. As a consequence, no communication event will ever take place out of order, implying that Lamport's Clock Conditions are met at any time which, by that, guarantees the causality within the simulation [75]. Hence, there is no need for so-called PDES (Parallel Discrete-Event Simulation) algorithms, like the ones discussed in [43, 91, 60], to synchronise the distributed simulation clocks. This is in contrast to, for instance, the WWT which uses a conservative PDES algorithm to guarantee causality [117, 96].

When applying the above parallelisation scheme, it may seem that the (sequential) communication model could become a potential bottleneck. For two reasons, however, this is unlikely to occur. First, the number of communication requests typically is much smaller than the number of computational operations. This is especially true for a large and important class of multicomputer applications, namely that of scientific (computationally intensive) applications. For applications which are constrained by communication (which will probably not run very well on multicomputers anyway), the communication model may indeed limit the parallel simulation performance. Despite the fact that communication is not extremely dominant in our applications, the communication model is still essential for cor-

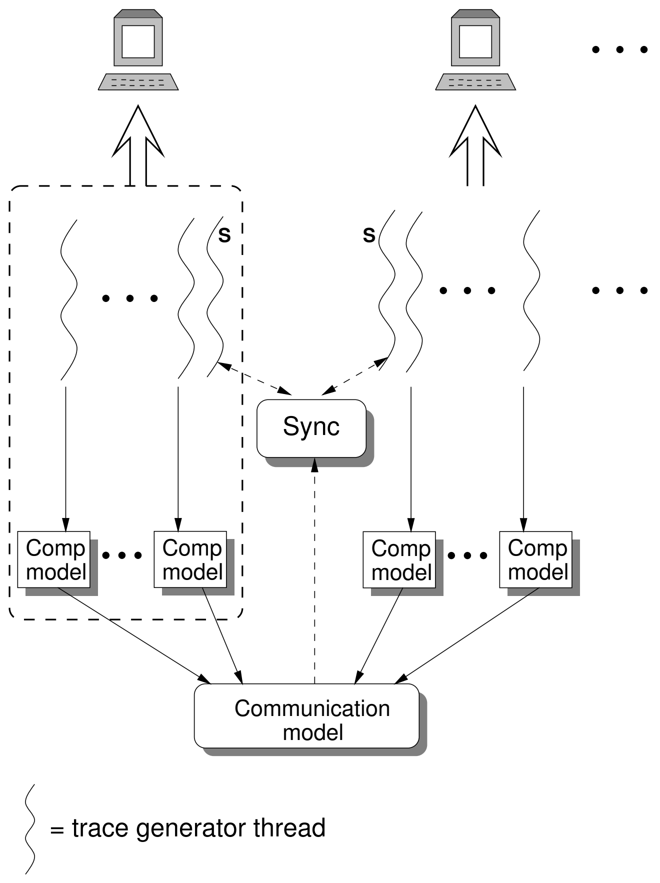


Figure 4.3: Distributed multicomputer simulation with Mermaid.

rectly modelling the synchronisation behaviour of the applications. Second, Section 3.1.2 explained that many multicomputer applications do not require execution-driven simulation. In such cases, the communication model can operate in pure trace-driven mode, thereby not constraining the execution of the threaded trace generator.

The question that remains is whether or not to parallelise the threaded trace generator. We decided to perform its parallelisation for reasons of scalability. By dividing the generator threads over multiple workstations, the threads' memory requirements are spread over multiple machines as well. This allows for scaling the simulation to larger applications and to target architectures containing more processors.

The distributed version of the hybrid simulation model is shown in Figure 4.3. The trace-generating threads are, together with the instances of the computational model which they feed, spread over multiple workstations. This division of work can be performed according to any distribution scheme. But here we assume that the work is evenly shared among all the hosts. As was mentioned earlier, the communication model is not parallelised and is therefore executed on a single host machine.

A special process, referred to as SYNC, coordinates the synchronisation of the distributed threads of the trace generator when simulator feedback is needed or when synchronisations at the application level have to be performed (see Section 3.1.2). In the case execution-driven simulation of the communication model is required, the SYNC process provides the threaded trace generator with the feedback from the architectural simulator. To send the simulator feedback to the appropriate thread, SYNC is connected to all participating workstations. On the other hand, if the communication model does not need execution-driven simulation, then SYNC can perform synchronisations at the application level, so directly between the trace-generating threads. To do so, remote threads are able to receive and send messages from/to each other via SYNC to synchronise and to exchange data in order to keep their notion of local data consistent. Remember, however, what was already explained in Section 3.1.2 that if the workload execution is not dependent on the transferred data, then SYNC does not have to perform the data transfers (nor any synchronisations) at all.

To coordinate these types of control at the side of the trace generator, each distributed part of the trace generator has one extra thread, called the *S-thread*, which takes care of the communication between the trace generator and SYNC. The S-thread will, for instance, signal (and possibly wake up) a trace-generating thread when data for it has arrived from another, distant thread.

Typically, the SYNC process is placed onto the same host as the communication model. Communication between the different components within this distributed environment is performed by either Unix sockets or shared memory, depending on the location of the communicating processes. For example, the threaded trace generator communicates via shared memory (or pipes) with the computational model, whereas each instance of the computational model uses a socket to talk to the communication model.

4.3.1 Parallel simulation performance

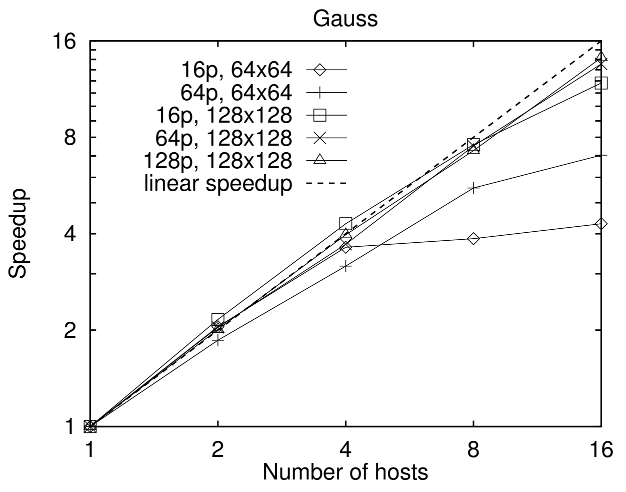
To evaluate the performance of parallel Mermaid, we used the multicomputer architecture model from Section 4.1. Again, we simulated the operation-traces of the three benchmark applications *gauss*, *pdmm* and *sort* (see Table 4.3). As was mentioned earlier, none of the benchmarks requires execution-driven simulation and only the execution of *sort* is data dependent. The latter implies that, when simulating *sort*, the SYNC process transfers data between remote threads to keep their notion of local data consistent. In the case of the other two benchmarks, SYNC is simply not used.

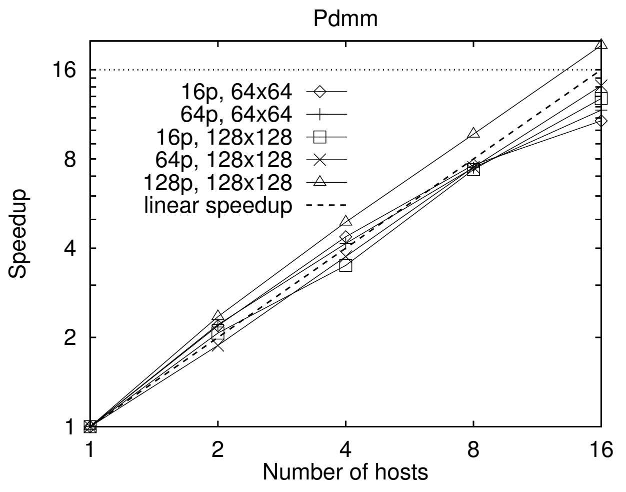
The experiments were performed using multicomputer configurations of 16 processors (*16p*), 64 processors (*64p*) and, where possible, 128 processors (*128p*). The cluster of host workstations over which the simulation is distributed consists of sixteen 110Mhz Sun Sparc-4s connected by normal Ethernet. These are not particularly high-end machines, but they form a lightly loaded, homogeneous cluster of workstations. In the future, we might also build a parallel version of Mermaid for a more tightly-coupled, distributed-memory MIMD platform.

Figure 4.4 shows for each benchmark the speedups of the parallel simulation environment. These measurements were performed by using wallclock times. For the parallel simulations, this includes both the actual simulation time and the time it takes to distribute the processes over the multiple hosts. Note that both axis have a logarithmic scale.

Gauss

The graph at the top of Figure 4.4 presents the results for *gauss*. These results clearly indicate that most of the obtained speedups are substantial. For instance, the *128p* configuration with 128$\times$128 matrices is simulated 14.2 times faster on 16 hosts than it is simulated on a





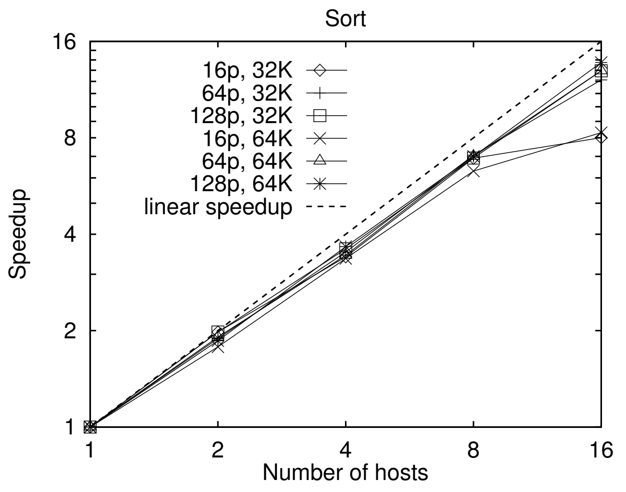


Figure 4.4: Performance results of parallel Mermaid.

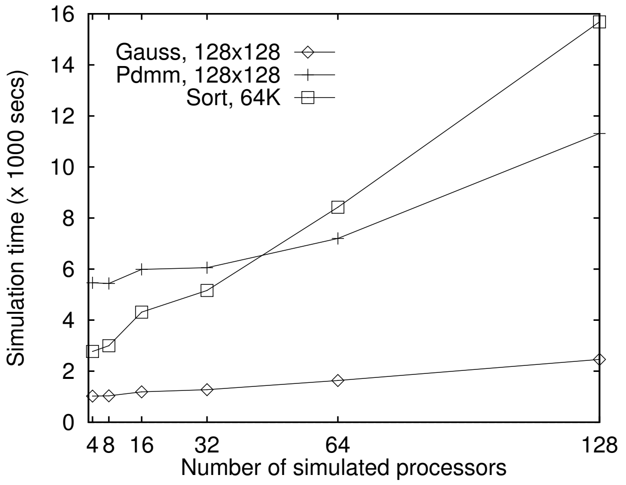


Figure 4.5: Simulation time v.s. the size of the simulated multicomputer on a single host.

single host. For nearly all simulations using two hosts we measured a super-linear speedup. This is most probably due to caching effects. Only the simulations using 64×64 matrices fail to obtain significant speedups beyond 4 host platforms. Apparently, the grainsize of these workloads is too small.

The fact that the multicomputer configurations containing more processors perform better than those with fewer processors is caused by the increased overheads in the simulations of larger configurations. This will be elaborated upon in the discussion of *pdmm*'s results.

Pdmm

For *pdmm*, of which the results are shown in the middle graph of Figure 4.4, the parallel performance is even better. All simulations properly scale with the increasing number of hosts. Using 16 hosts, for example, speedups of between 10.75 and 19.3 are obtained. Thus, again super-linear speedups were measured. In fact, all parallel *pdmm* simulations of the *128p* configuration with matrices of 128×128 obtain super-linear speedups. To explain this, consider Figure 4.5. For each benchmark, this graph shows the simulation time as a function of the number of simulated processors on a single host. The graph demonstrates that the simulation time of all benchmarks is at least doubled when the multicomputer simulation is scaled up from 4 to 128 processors. This is caused by the increase of overheads that are related to the threaded execution of the sequential simulation, such as the thread-scheduling overhead and the locking overhead. Since the overheads per host within the distributed simulation are smaller (i.e. the overheads are parallelised), super-linear speedups might be obtained. In the case of *pdmm*, the simulation time for a single host starts to increase significantly after 64 processors. This corresponds with the super-linear speedup of *128p* in Figure 4.4. The attained super-linear speedups also indicate that the implementation of the sequential simulation may be improved. Perhaps a more efficient thread-package (Sun's Solaris threads are currently used) reduces the thread overhead such that super-linear speedups are not encountered anymore.

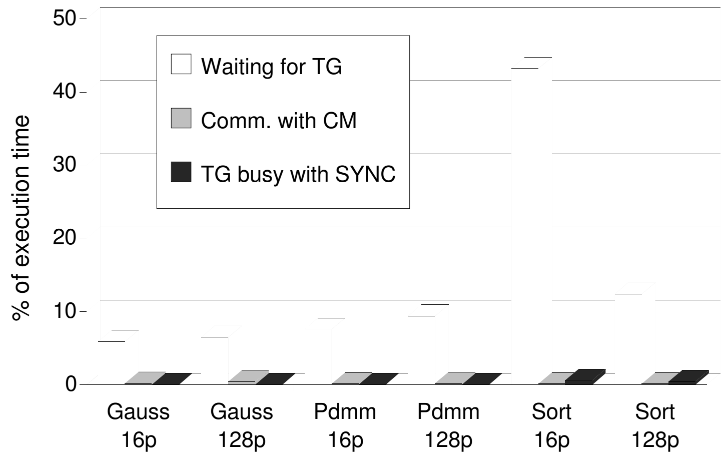


Figure 4.6: Breakdown of where the execution time goes due to certain parallel overheads. The white bar shows the average time the computational model waits for the Trace Generator (TG), the grey bar indicates the average time the computational model communicates with the Communication Model (CM). Finally, the black bar represents the average time the TG is busy transferring data via the SYNC process. All numbers are percentages of the total execution time. These experiments were performed for 128x128 matrices (gauss and pdmm) and 64K of integers (sort) on a cluster of 16 hosts.

Sort

The graph at the bottom of Figure 4.4 shows the results of *sort*. Again, substantial speedups are obtained. Although Figure 4.5 shows an impressive increase of simulation time for larger multicomputer configurations, no super-linear speedups were measured for *sort*. This is due to the synchronisations between the remote trace-generating threads, which cover up the gain of parallelising the overheads. To illustrate this, consider Figure 4.6. This graph shows a breakdown of the average overheads in the parallel instances of the trace generator and the computational model. These results are for the *16p* and *128p* simulations of all three benchmarks on 16 hosts with the largest data sizes. Three types of parallel overheads are distinguished: the time the computational model waits for operations from the trace generator (white bars), the time the computational model communicates with the communication model (grey bars) and the time the trace generator is busy with transferring data via the SYNC process (black bars). Note that all three types of bars are overheads and do therefore not sum up to 100%. The white bars in Figure 4.6 demonstrate that, in the case of *sort*, the computational model generally waits longer for the trace generator compared to the other two benchmarks. This can be explained by the fact that a trace-generating thread may be suspended (i.e. waiting for remote data) for a quite a while before being resumed by the remote thread. These synchronisation overheads do not occur in the other two benchmarks, as they do not perform remote data transfers. Nevertheless, it may seem that the difference in overhead between the simulations of, for example, *pdmm*'s and *sort*'s *128p* configuration is only marginal. Note, however, that the results are averages and we measured a standard deviation (σ) of 3.4% for *sort*, while *pdmm*'s σ only equals to 0.2%. The large σ for *sort* is caused by a few parallel instances suffering from high synchronisation overheads of which the largest equals to 22% of the execution time.

Figure 4.6 shows that the synchronisation overhead is extremely high for the *16p* simulation of *sort*: the computational model waits on the average for more than 40% of the execution time for the trace generator. So, this must the reason for the poor parallel performance of the *16p* simulations on 16 hosts (see Figure 4.4). It is, of course, not surprising that this particular configuration suffers from these overheads as there resides only one simulated processor on a host. Thus, when suspending a thread, no other thread can take over and the simulation on the host is entirely stalled.

Due to the higher, parallelisable, overheads of sequential simulation of larger configurations and the existence of multiple trace-generating threads per host, the performance of the *64p* and *128p* simulations does scale up after 8 host machines.

4.3.2 Discussion

For the simulated benchmarks, the results indicate that the sequential communication model does not constrain the parallel simulation performance. As the grey bars in Figure 4.6 show, the overhead of the communication between the computational model and the communication model is for all benchmarks exceptionally small. Figure 4.6 also demonstrates that the overhead of data transfers by SYNC is neglectable (black bars). Naturally, this overhead only exists for *sort* as the other two benchmarks do not use SYNC. Note that, in this study, all necessary synchronisations within the trace generator are performed at the application level rather than at the architecture level. We still need to investigate how architectural feedback would affect the parallel simulation performance.

The obtained performance improvements due to our parallel extensions are very promising. Whereas the original Mermaid simulators obtain a typical slowdown of between 60 and 650 per processor compared to the real machine, parallel Mermaid may reduce these slowdowns by an order of magnitude. The average speedup, for instance, for all performed simulations with 16 host computers is nearly 12. So, when reconsidering Table 4.4 and assuming that the slowdown now is in the range of 6 to 65, it is clear that we obtain one of best slowdown figures. Only the Wisconsin Wind Tunnel [117, 96], being a direct-execution distributed simulator, yields better simulation performance.

Furthermore, the obtained super-linear speedups illustrate the improved scalability of parallel Mermaid with respect to its sequential counterpart. Simulating large multicomputer configurations on a single host machine may easily lead to performance degradations due to increased overheads or is simply impossible due to the excessive memory consumption of the workloads.

Up to now, the experiments were done using a homogeneous pool of 16 workstations. It would be interesting to investigate whether or not parallel Mermaid scales up beyond these 16 hosts. Moreover, in practice, heterogeneous workstations with different computational powers might be used. This would accentuate the importance of load-balancing strategies to optimally divide the computations over the hosts.

4.4 Summary

In this chapter, we presented a validation and efficiency study of the Mermaid simulation methodology. The validation experiments show that good accuracy can be obtained. For the multicomputer model under investigation and the applied benchmarks, we generally measured small errors between the real execution time and the predicted one. On the average, these errors do not exceed 5% with only modest variance.

Besides Mermaid's good accuracy, its simulators are fairly efficient as well. For the experiments we performed with a simulator running on a single host computer, we measured a typical slowdown of about 60 to 650 per simulated processor. To improve this performance, and to increase the scalability of the architectural simulator, we extended the simulation environment to allow distributed simulation on a cluster of host workstations. Experiments have shown that this parallel version of Mermaid obtains significant speedups reducing the original slowdowns by roughly an order of magnitude. Evidently, this performance makes Mermaid extremely competitive with, and in most cases faster than, many other efficient architecture simulators.

The set of benchmarks used in this chapter is rather limited. In the future, we therefore need to extend the benchmark suite. Doing so, we should definitely add non-deterministic applications to investigate the impact of workloads that require execution-driven simulation.

Part II

Case studies

Chapter 5

Evaluation of a Mesh of Clos network[†]

"Communication is not only the essence of being human, but also a vital property of life."

John A. Piece

The type of multicomputer network and its performance are an essential factor of the platform's total performance. Basically, two main types of networks can be distinguished. In *direct* networks, each node has a direct, or point-to-point, communication channel to a number of other nodes, called *neighbours*. Neighbouring nodes may send messages to each other directly, while messages between non-neighbouring nodes have to be routed by other nodes in the network [37]. The routing is either done by the processor itself or by a special-purpose routing device residing at the node. In *indirect* networks, there are no point-to-point connections between nodes. Instead, a node is connected to one or more separate routing devices, which on their turn can be connected to other routers. Consequently, all message passing requires routing by at least one of the routing devices. Example topologies for direct networks are 2D grids and hypercubes, while multistage topologies can be regarded as examples of indirect networks.

An important challenge in the development of networks is to design a topology which is affordable with respect to its wiring complexity and still has good topological characteristics. With the latter we mean, for instance, a small *network diameter* (the maximum shortest path between two nodes) and a high *bisection bandwidth* (the minimum bandwidth between all possible network halves). For a more comprehensive overview of these network characteristics, see for example [143].

Another important issue regarding the network performance is its message passing efficiency. This efficiency depends heavily on the process of transferring data from one node to another, commonly referred to as *switching* [123, 31, 68, 93]. In the past few years, one switching technique in particular, called *wormhole routing* [31], has become increasingly popular. The reason for this is that wormhole routing offers a low network latency while minimising hardware expenses.

This chapter presents a case study on the simulation and the performance evaluation of a wormhole-routed network connected in a so-called Mesh of Clos topology. Our interest

[†]This chapter is based on [107].

in this particular type of network originates from a series of Parsytec multicomputers that is based on this communication technology. Before the actual realisation of these machines, we evaluated their potential network performance using the Mermaid modelling framework.

5.1 Wormhole routing

Early multicomputers used store-and-forward switching [101]. In this switching technique, when a message reaches an intermediate node, the entire message is stored in a message buffer. The message is then forwarded to a selected neighbouring node when the appropriate output channel is available and the destination node has an available buffer. This method has two drawbacks. First, each node must buffer every incoming message, consuming quite a large amount of memory space. Second, the network latency is, besides the packet size, also proportional to the distance between the source and destination nodes.

To overcome these problems, a new switching technique, called *wormhole routing* [31], was introduced. In this technique, a message is divided into a sequence of small, constant-size *flits* (flow control digits). The first flit (the header) of the sequence holds the destination's address since it is used to determine the path the message must take. As the header advances along its route, the trailing flits follow in a pipelined fashion. This results in a network latency that is nearly distance insensitive if there exists no channel contention between messages [101]. To illustrate this, consider Figure 5.1 in which the difference in latency of wormhole routing and store-and-forward switching is shown.

In wormhole routing, once a channel has been acquired by a message, the channel stays reserved until the last flit (the tail) has been transmitted. Whenever the header encounters a channel that is already occupied by another message, the headerflit is blocked until the channel in question is released. Instead of being buffered in one large message buffer, the trailing flits stay in the network during the stall of the header. They are stored in small flitbuffers along the established route, which eliminates the need for large message buffers at intermediate nodes. The fact that channels stay reserved during a message-stall has one potential disadvantage. The idle, but reserved, channels cannot be used by other messages which may for this reason also block, as illustrated in Figure 5.2. This can lead to a snowball-effect, causing so-called *tree-saturation*.

By introducing *virtual channels* [30], which are multiplexed onto the physical channel, this problem can be solved. Each virtual channel has its own flit-buffers which allows mul-

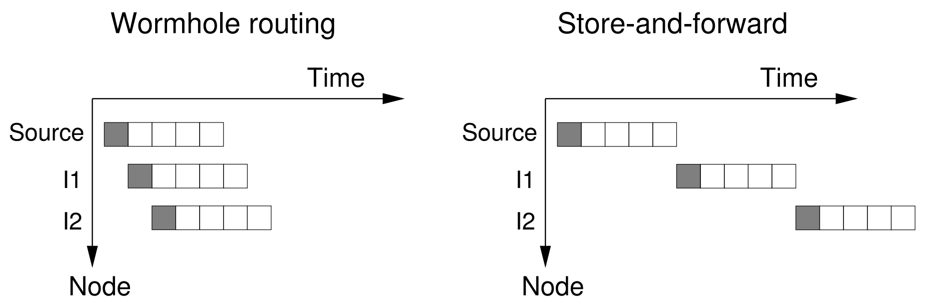


Figure 5.1: The difference in latency of wormhole routing and traditional store-and-forward switching. The grey parts denote the message header.

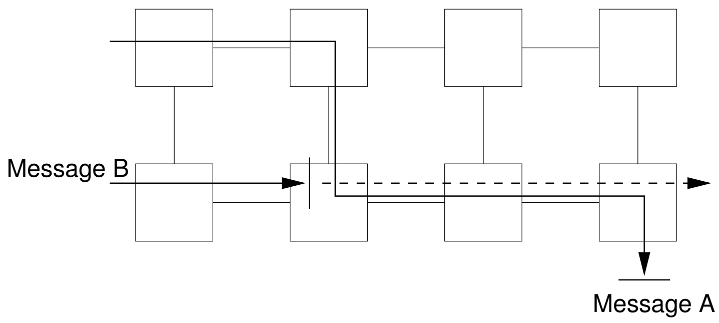


Figure 5.2: A channel conflict. Message A is blocked and keeps its channels reserved. Message B cannot use A's idle, but reserved, channels and is therefore also blocked.

tiple message transfers to be active on a channel at the same time. In this study, however, we did not apply the concept of virtual channels. So, here we only focus on pure wormhole routing.

5.2 The Mesh of Clos topology

Increasing the efficiency of communication networks by changing their topology often leads to a decrease of realisability due to the high cost. Particularly, networks with multistage topologies can offer a small diameter, large bisection width and a large number of redundant paths but are hard and expensive to construct because of the complex wiring structure. By combining a multistage network with an easy to realise mesh network, the Mesh of Clos network [92] addresses this tradeoff between efficiency and realisability. As its name already suggests, it is based on the Clos multistage network [24]. We define a Clos network of height h, constructed of routers with $2k$ bidirectional communication channels, by the following recursive scheme:

- A single router with $2k$ bidirectional communication channels of which k are connected to nodes is a Clos network of height 1.
- A Clos network of height h is built by connecting k Clos networks of height $h-1$ by k^{h-1} routers. Since each of the k subnetworks has k^{h-1} external channels, k^{h-1} routers are used at level h such that the i-th external channel of each subnetwork connects to the i-th router at level h.

Figure 5.3 shows a Clos network of height 2 in which four channels of each router at the top level are left unused. Subsequently, the Mesh of Clos(h,r) topology, or in short *MoC(h,r)*, is defined by replacing the r top stages of a Clos network of height h by a $2^r \times 2^r$ mesh structure. Here we assume that the routers are configured with eight communication channels of which at most four can be connected to nodes, i.e. $k = 4$. Two examples of a Mesh of Clos network are depicted in Figure 5.4. Both are configured in a 2×2 mesh, the first having *clusters* of four nodes and the other having clusters of sixteen nodes. The latter is actually called a *Fat Mesh of Clos* since the mesh structure that interconnects the clusters can be regarded as four independent layers of 2D meshes (as illustrated by the different shaded surfaces in Figure 5.4). Note that there are several optimisations possible for the networks

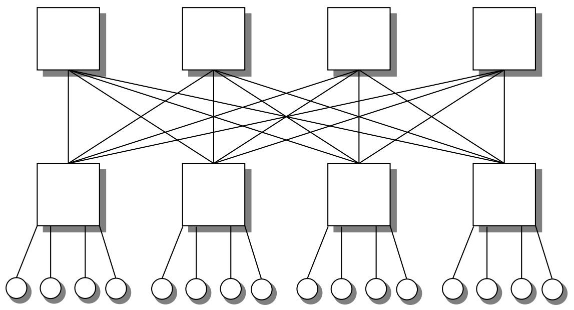


Figure 5.3: A Clos network of height 2 with $k = 4$. Boxes refer to routers and circles to nodes. All channels are bidirectional.

shown in Figure 5.4 by connecting unused channels. However, as these optimisations are not generically applicable, we do not consider them and only focus on the pure Mesh of Clos topology.

Table 5.1 gives a comparison of the network parameters for the Mesh of Clos and mesh topologies. From this table can be seen that, for small values of r, the Mesh of Clos has better network characteristics than a mesh. Another important issue is, of course, the number of routers (and channels) the network requires to be built. Evidently, a mesh containing N nodes requires N routers. The number of routers in a Mesh of Clos(h,r) with N nodes and $k = 4$ is given by the equation

$$\text{Routers}_{MoC} = \begin{cases} 0 & \text{if } h - r \leq 0 \\ ((\frac{1}{2}\log_2 N) - r) * 4^{(\frac{1}{2}\log_2 N) - 1} & \text{otherwise} \end{cases} \tag{5.1}$$

Figure 5.5 draws this function (dark surface) against the number of routers required by a mesh network (light surface). The axis labeled with r defines the number of top stages that

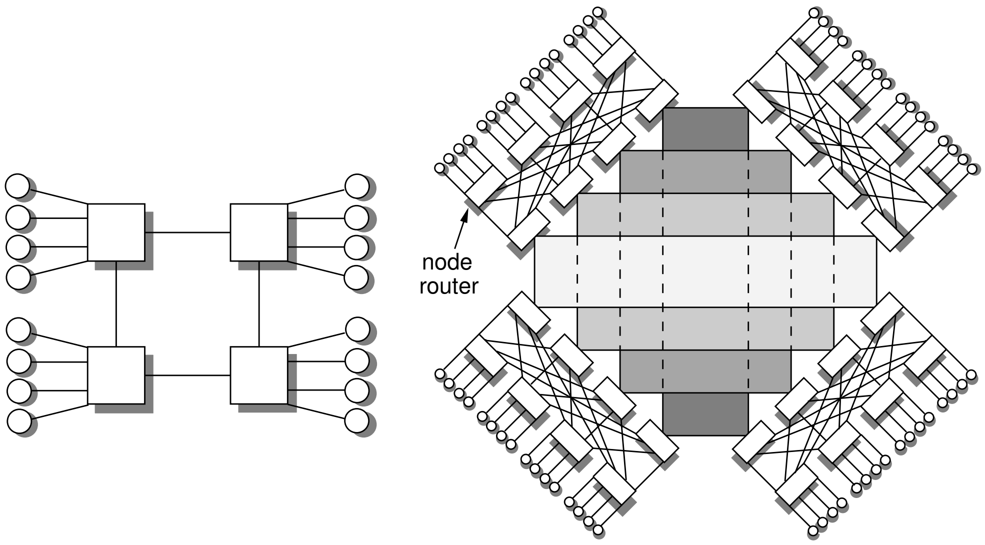


Figure 5.4: A Mesh of Clos(2,1) network (left) and a Mesh of Clos(3,1) network (right). The boxes refer to routers and the circles to nodes.

	Mesh of Clos(h,r)	Mesh
Router degree	8	4
Diameter	$2 \cdot (2^r - r - 1) + \log_2 N$	$2 \cdot (\sqrt{N} - 1)$
Bisection width	$N \cdot 2^{-r-2}$ (with $r > 0$)	$\sqrt{N}$

Table 5.1: Network parameters of the Mesh of Clos and mesh networks containing N nodes.

are removed from the Clos network to form the Mesh of Clos. Note that the surface within the triangle between $r = 2$ and 1024 nodes is not defined because r would then be equal to or larger than the height of the Clos network. Or, in other words, all Clos levels or more levels than there are available would be removed. Figure 5.5 illustrates that the number of routers, which is rapidly increasing for a pure Clos ($r = 0$), can be reduced considerably by replacing the top stages of the Clos for a mesh structure, i.e. by increasing r. However, as demonstrated by Table 5.1, r should not become too large in order to preserve a small network diameter and a large bisection width. More specifically, if $r \rightarrow h$ then the MoC's network diameter becomes similar to that of the mesh network while the bisection width may eventually become worse than that of the mesh.

Monien et al. have performed a study on the behaviour of several Mesh of Clos networks using an analytical model [92]. They found that for communication patterns exhibiting high locality only a slight decrease in performance is introduced with respect to a pure Clos network. For communication patterns that are communicating across the mesh network more frequently, the performance decrease will be larger due to the limitations of the mesh network.

In this evaluation study, we restrict us to *instances* of the two Mesh of Clos topologies shown in Figure 5.4. We scale the mesh parts of these Mesh of Clos networks while keeping their Clos parts untouched. The reason for this is that the machines under investigation are based on these topologies.

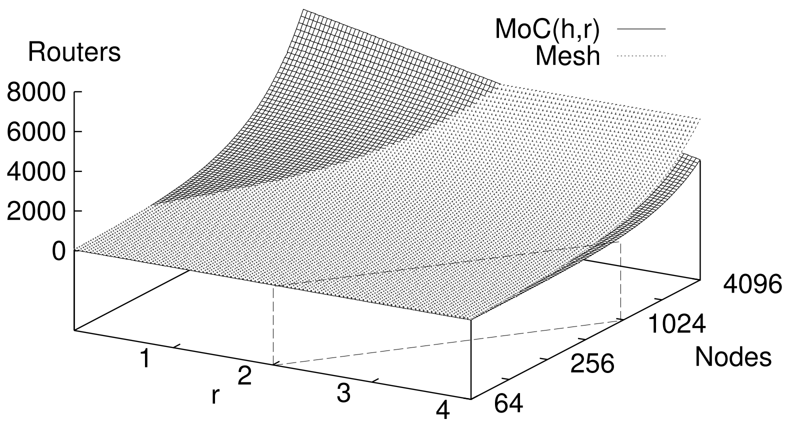


Figure 5.5: Number of routers required for a Mesh of Clos (dark surface) and a normal mesh network (light surface).

5.2.1 Routing

Although its name may suggest otherwise, wormhole routing is purely a switching technique and does not deal with the routing of messages (determining the path of messages through the network). Therefore, we also have to define how messages find their way from source to destination. Throughout this case study, we assume that routing of messages in the mesh structure is performed by deadlock free, deterministic XY routing [101]. In this technique, a message is first routed to the appropriate X-coordinate, then to its Y-coordinate. For the Mesh of Clos networks containing Clos structures of height 2, however, additional routing is required within the Clos parts. The routers directly connected to the nodes, called *node routers* (see Figure 5.4), must decide at which mesh-layer messages should travel to their destination. For now, we assume a deterministic approach in which a node always sends data over a pre-defined and fixed layer according to the following scheme: if a node router connects to nodes n_i (with $0 \leq i \leq 3$) and the four mesh-layers are numbered 0 to 3 then messages from node n_i are routed to mesh-layer i.

5.3 The simulation approach

To simulate the Mesh of Clos network, we have used Mermaid's communication model and changed it to represent the infrastructure as shown in Figure 5.6. This figure depicts the model of a four node cluster which contains all the basic components. The model, implemented by Pearl objects, consists of a processor, a Virtual Communication Processor (VCP), two channels (input and output) and a router. The bidirectional communication channels are modelled by two unidirectional channel objects. A channel object is a straightforward forwarding mechanism with a certain latency. The processor component generates the communication requests which are dispatched to the VCP. After receiving such a communication request, the VCP is responsible for handling the transmission. The message setup latency is split up and modelled within both the processor and VCP components. Finally, the router component routes all incoming flits according to a specified routing algorithm. It connects to other intermediate routers or to neighbouring clusters in order to realise the Mesh of Clos network.

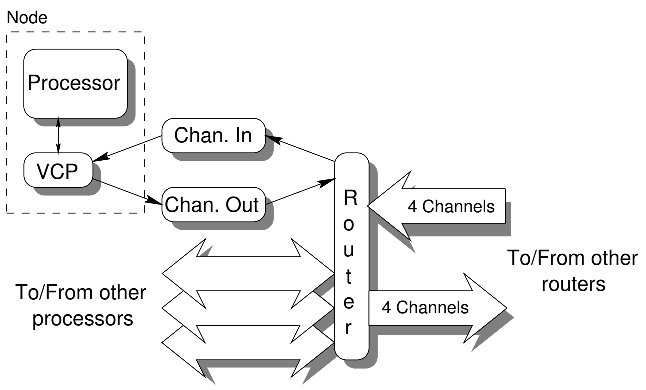


Figure 5.6: The simulation model of a cluster containing four nodes. An oval box refers to a basic model component.

5.3.1 Efficient wormhole routing simulation

An important issue in simulating large-scale multicomputer networks is to efficiently model the communication behaviour within the network. Explicit simulation of each separate flit must be avoided because this would result in a simulation time that is linear in the message size and in the distance traveled by the message. Instead, full advantage should be taken of the pipelined fashion in which flits move through the network. This behaviour allows for explicitly simulating the header and tail flits only. The movements of the intermediate flits are implicit. This approach basically results in a simulation time that is insensitive to the message size, and thus is only linear in the distance to travel.

Wormhole-routed networks sometimes use multiple flitbuffers per incoming channel on a router as this may have a positive effect on the communication performance. By increasing the buffer space, channels may be unblocked earlier, thereby potentially improving the throughput. However, the presence of multiple flitbuffers per channel on a router makes efficient simulation slightly more difficult: once the header flit is blocked, the trailing flits continue to be transferred while there are enough free flitbuffers. In [88], McKinley et al. present a simulation algorithm that exploits the implicit movement of intermediate flits and that solves the problem of multiple flitbuffers. The outline of this algorithm when applied to the transmission of a single message is shown in Figure 5.7. There are three variables defined per message. The first, *tts*, is an array containing the amount of time *tts[i]* that channel *i* requires in order to send the flits currently stored in the flitbuffers of the sending router. Each element of *tts* is associated with a router along the path followed by the message. The variable *blocked_at* indicates the amount of time until all movements of trailing flits stall due to full flitbuffers. Finally, *last_chan* refers to the channel longest held by the message. The key to efficiency for this algorithm is the use of a single variable (i.e. *tts*) representing the state of a message with regard to each router it traverses.

Our network simulator has been implemented in both the naive manner (simulating every single flit) and using an optimised algorithm which is derived from the one discussed in [88]. The optimised algorithm, as sketched in Figure 5.7, was implemented partly in C and Pearl and has been embedded in the VCP and channel components of our model (see Figure 5.6). We used the naive model to verify the more sophisticated implementation of the optimised model with small communication loads. The efficient simulation model showed an efficiency improvement of more than an order of magnitude compared to the naive approach.

5.4 Experiments

To evaluate the Mesh of Clos network, four types of *synthetic* communication loads were used. The loads perform synchronous communication, which means that every message must be explicitly acknowledged. As a result, the acknowledgements put an additional load onto the network. A side-effect of the synchronous communication is that the workloads are self-regulating, implying that the rate at which messages are issued to the network depends on the amount of network traffic (i.e. the network latency). When the network latency is high, the time between the issuing of two messages will increase as it takes longer before the first message is acknowledged. The reason for using synchronous communication origi-

```
tts[i]      : time to send in order to free all buffers
              at channel i.
blocked_at  : time to send until transmission of all flits
              within the message is stalled due to full
              buffers.
last_chan   : longest held channel by the message.

/* “advance tailflit” means releasing last_chan   */
/* and updating last_chan with the next channel */
/* along the route                                */

while the headerflit has not arrived at destination do
  route headerflit along channel next_chan
  while next_chan is occupied by other message do
    if tts[last_chan] $<$ blocked_at then
      simulate time until tts[last_chan] time units
      have past or next_chan has been released
      if next_chan has not been released then
        advance tailflit
      fi
    else
      simulate time until blocked_at time units have
      past or next_chan has been released
    fi
    update tts array and recompute blocked_at
  done
  if tts[last_chan] $\leq$ transmission time of a flit then
    simulate transmission time of a flit, advancing
    tailflit after tts[last_chan] time units
  else
    simulate transmission time of a flit
  fi
  update tts array and recompute blocked_at
done
while tailflit has not arrived at destination do
  simulate tts[last_chan] time units
  advance tailflit and update tts array
done
```

Figure 5.7: Algorithm for efficient simulation of wormhole routing.

	# Nodes	Mesh size	Clos height	Diameter	Bisection width	# Routers
8×8 mesh	64	8×8	—	14	8	64
MoC(3,2)	64	4×4	1	8	4	16
MoC(3,1)	64	2×2	2	6	8	32
16×16 mesh	256	16×16	—	30	16	256
MoC(4,3)	256	8×8	1	16	8	64
MoC(4,2)	256	4×4	2	10	16	128

Table 5.2: Network characteristics of the modelled networks.

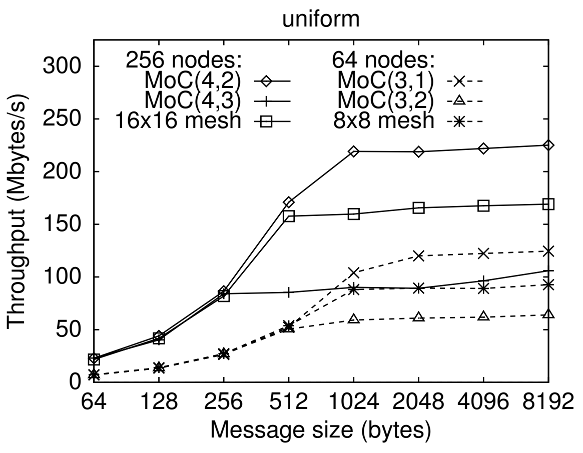
nates from the fact that this type of communication is widely used in multicomputer applications. For example, most SPMD programs communicate via synchronous message passing.

In the first communication load, called *uniform*, messages are sent in a random manner such that the destinations are uniformly distributed. The second load, referred to as *hotspot*, represents a less ideal model of communication. It defines several non-neighbouring nodes that receive a disproportionately large number of messages, causing hotspots to appear in the network. This type of load is similar to the one used by Pfister and Norton to study hotspots in shared memory systems [106]. The third load, called *hotregion*, distinguishes a cluster of 12 nodes (for the smaller networks) or 16 nodes (for the larger networks) forming a "hot region", rather than defining single nodes as hotspots. In both *hotspot* and *hotregion*, the total hotspot area receives 40% of all messages. Finally, *partner* represents a point-to-point load, in which every processor communicates with one fixed partner. The partner tuples are formed such that there exists a variety of routing-path distances and there is a relatively high degree of channel contention. For all communication loads, messages are generated at fixed intervals of time.

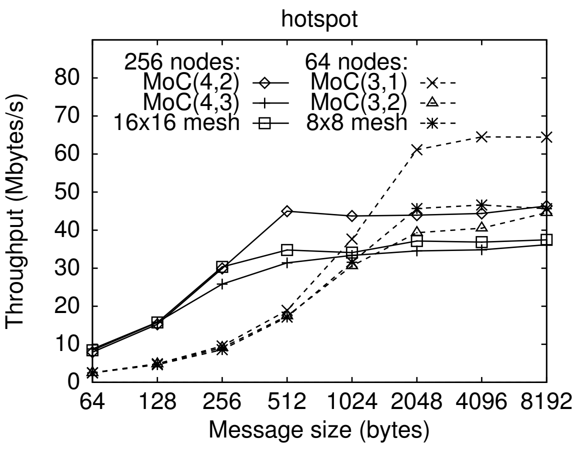
The communication loads have been simulated for four different Mesh of Clos topologies. Two of them are based on Clos structures of height 1, whereas the other two contain Clos structures of height 2. The Mesh of Clos networks of height 1, the MoC(3,2) and MoC(4,3), are similar to the MoC(2,1) network shown in Figure 5.4. However, they contain meshes of 4×4 and 8×8 respectively. The Mesh of Clos networks of height 2 are the MoC(3,1) (see Figure 5.4) and the MoC(4,2), of which the latter contains a 4×4 mesh. To compare the Mesh of Clos results, we have also simulated the traditional and widely-used mesh topology using the same model components as our Mesh of Clos model. Table 5.2 shows an overview of the characteristics of the modelled networks.

Message setup time	70 μs
Channel bandwidth	40 Mbyte/s
Routing overhead	100 ns/flit
Packet creation overhead	2.5 μs
Memory latency per word	100 ns

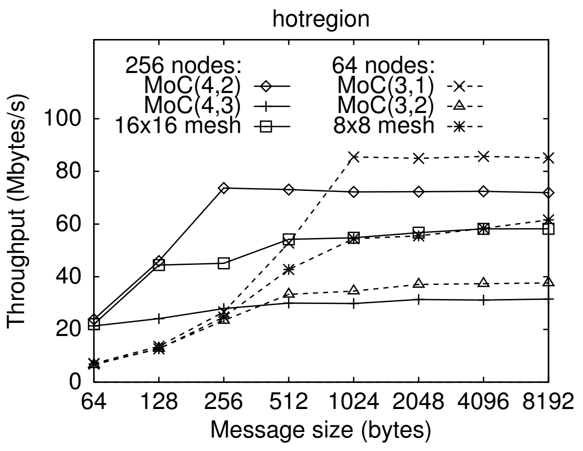
Table 5.3: Communication parameters for all network models.



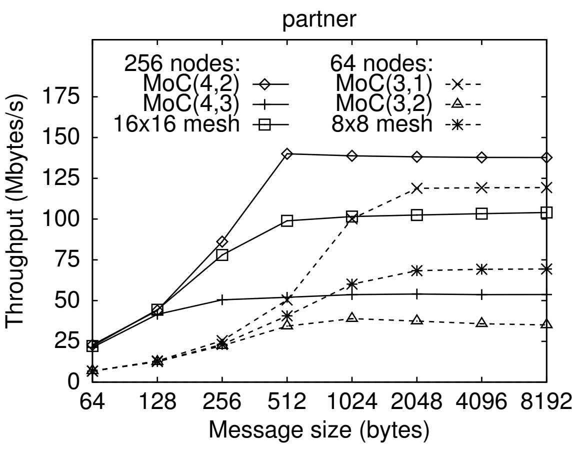
(a)



(b)



(c)



(d)

Figure 5.8: Estimated throughputs for the *uniform* (a), *hotspot* (b), *hotregion* (c) and *partner* (d) communication loads.

Every network model is parameterised with the same latencies which were derived from the specification of an early model of the multicomputers under investigation. These communication parameters are summarised in Table 5.3. By default, the routing devices are equipped with one flitbuffer per incoming channel. Furthermore, the very limited operating system functionality that is modelled on the nodes takes care of splitting up messages into packets of 128 bytes. After this is done, the packets are divided into single byte flits with a header of 2 flits attached to each packet.

Figure 5.8 shows the estimated total network throughputs for the different types of communication loads. The solid lines refer to the networks containing 256 nodes, whereas the dotted lines refer to the networks of 64 nodes. The graphs do not give the raw network performance since the effects of message setup overheads and acknowledgement overheads are also included in the results.

The point after which no additional throughput is obtained for increasing message sizes, indicates that the network is saturated. For the *hotspot* and *hotregion* loads, Figure 5.8 shows that the networks containing 256 nodes reach this point of saturation earlier than their counterparts of 64 nodes. The reason for this is that the larger number of nodes can stress the specific hotspot areas more intensively. The graphs clearly show the decrease of throughput when communication is not distributed uniformly. For example, the hotspots created by the *hotspot* workload lead in many cases to a throughput deterioration of more than 50% compared to uniform communication.

The MoC(4,2) and MoC(3,1) networks, being the Mesh of Clos networks of height 2, achieve the highest throughputs for all communication loads. Using these topologies, we obtained throughputs that are calculated to be 75% higher than those for the normal mesh network. This suggests that these Mesh of Clos networks are less prone to contention than the more flat networks that have been examined. As a consequence, the Mesh of Clos networks of height 2 often saturate more slowly than the other networks. To illustrate this, consider the graph of *uniform* in Figure 5.8. It shows that the 16×16 mesh saturates at a message size of 512 bytes, while the MoC(4,2) reaches the point of saturation at a message size of 1Kb.

Interestingly enough, the results also indicate that the performance of the mesh networks is superior to that of the Mesh of Clos networks of height 1 (i.e. the MoC(4,3) and MoC(3,2) networks). Apparently, the routers within the latter type of network suffer from high contention. This can be explained by the small bisection width of the MoC(4,3) and MoC(3,2) networks. The routers in these networks must handle traffic from both their four nodes and from their neighbouring routers within the flat mesh structure. As the Mesh of Clos of height 1 shows such poor performance, we will not use this type of network for further evaluation. The next sections, which address the buffering and routing characteristics of Mesh of Clos networks, will therefore only focus on the MoC(4,2) topology.

5.4.1 Multiple flitbuffers

To investigate the influence of increasing the number of flitbuffers per channel, we simulated the communication loads using various router configurations. In these experiments, each applied flitbuffer contains one byte as we use single-byte flits. Figures 5.9 and 5.10 display the relative increase of throughput for several message sizes due to multiple flitbuffers in a MoC(4,2) network. It shows that many communication loads modestly benefit from a larger buffer space. The highest measured gain of throughput equals to 14%. Naturally, the larger number of flitbuffers has the greatest impact on the *hotspot* and *hotregion* loads, as their hotspot areas cause high contention. For small message sizes (i.e. 128 bytes), none of the communication loads does improve. This is because not enough network traffic is generated to cause the contention that is needed to benefit from the extra flitbuffers.

As can be seen from Figure 5.9 and 5.10, the performance results of the multi-flitbuffer configurations are less predictable than one would expect. For some instances of the communication loads, we even measured a decrease of throughput when adding flitbuffers. This unexpected behaviour is due to a number of effects. The multiple flitbuffers cause channels to be unblocked earlier, enabling new packets to enter the network and thereby increasing the network traffic. Subsequently, the header stall delays that packets encounter within the network will change due to the effects of the extra network traffic and the enlarged buffer space: packet headers are potentially more often blocked because of the higher network traffic, but have a shorter mean stall time due to the larger number of flitbuffers. This change

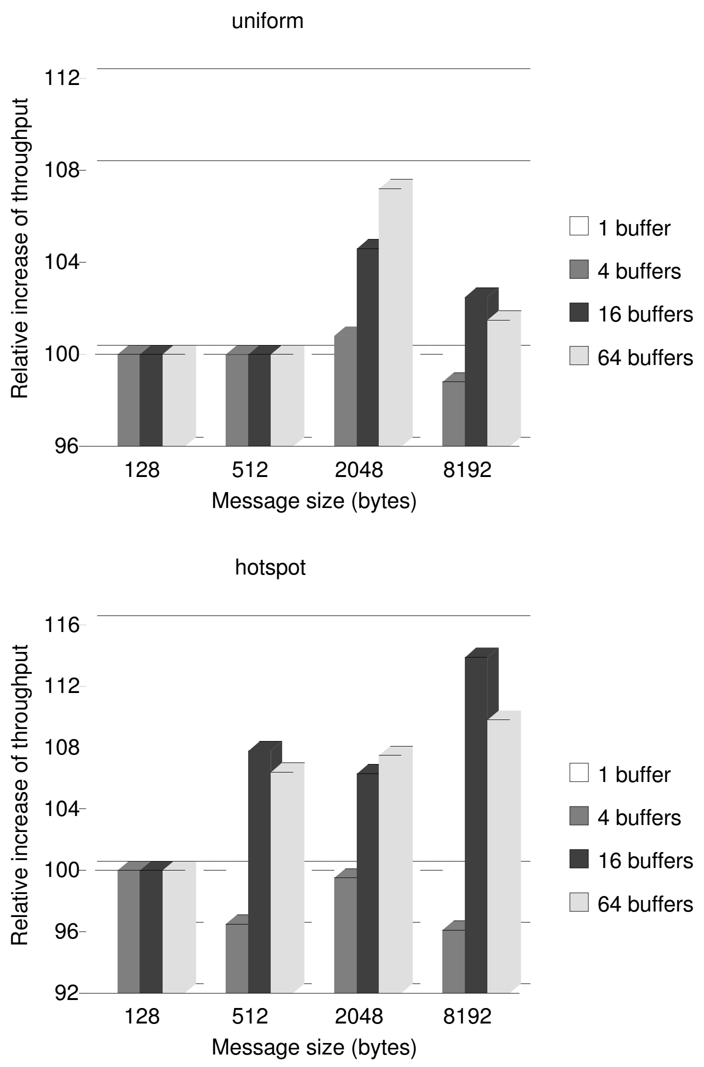
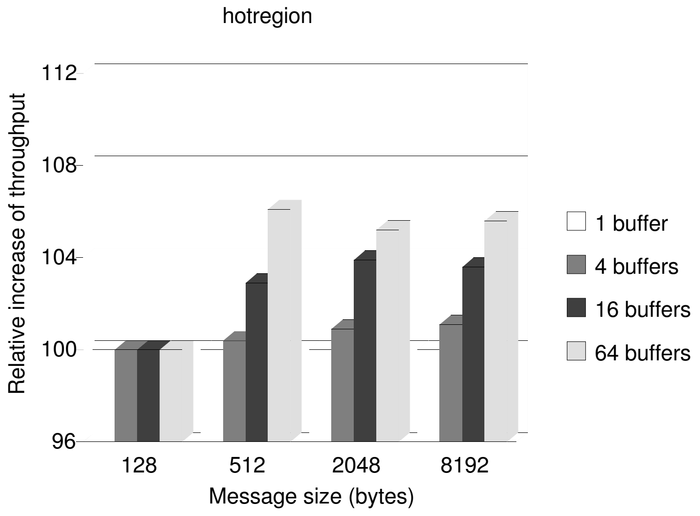


Figure 5.9: Relative increase of throughput due to multiple flitbuffers for *uniform* (top) and *hotspot* (bottom).

of header stall delays means that the order in which packets arrive at and are handled by routers will change as well. If such a change of packet handling order results in a delay of the packets that are on the critical path of the application load, then the overall throughput may decrease. This critical path can be formed by synchronisations, either by acknowledgements or at the application level.

To illustrate the above, consider Figure 5.11. It shows a routing scenario at two routers *R1* and *R2* in which a packet *A* has to be routed in western direction, while two other packets *B* and *C* need to be routed to the north. The packets *A* and *B* will contend for the eastern input channel at *R1*, and the packets *B* and *C* will contend for the northern output channel at *R2*. It is assumed that router *R1* routes packet *A* first and that packet *C* is on the critical path of the application. The latter implies that the transmission of packet *C* forms the critical path delay in this example. This scenario allows two different packet handling orders. If, due to the lack of free flitbuffers, packet *B* is still blocked before router *R1*, then *R2* routes packet *C* first. After this packet has left *R2*, packet *B* may be routed at the moment enough flitbuffers



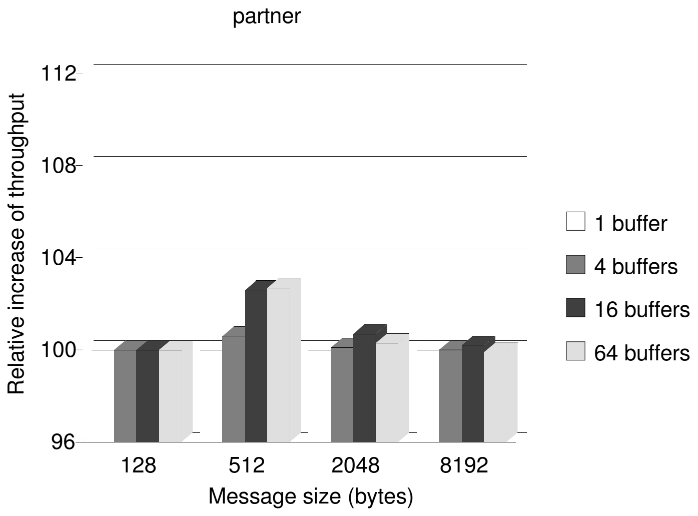


Figure 5.10: Relative increase of throughput due to multiple flitbuffers for *hotregion* (top) and *partner* (bottom).

become free. In this scheme, packet *B* does not interfere with the packets on the critical path (packet *C* in this example). On the other hand, if we add more flitbuffers at the routers then this would allow packet *B* to arrive first at router *R2*. In that case, packet *C* has to wait for packet *B* and is blocked at router *R2*. Now, the critical path delay has been increased by the transmission delay of packet *B* from router *R2*. Hence, this order of packet handling will result in a decrease of overall throughput.

As communication in our loads is synchronous, the synchronisations at the operating system level form an essential part of the critical path: a node cannot continue sending until it has received the acknowledgment of the previous message. To demonstrate the influence of changes in the order of packet handling on these synchronisations, the *hotspot* load will be subject to closer examination (as this load shows the most irregular behaviour). For *hotspot* using a message size of 512 bytes, Figure 5.12 shows the distribution of the cumulative time that nodes have been waiting for acknowledgements, which we call the *acknowledgement latency*. The Y-axis shows the number of nodes (i.e. the frequency) at which a particular

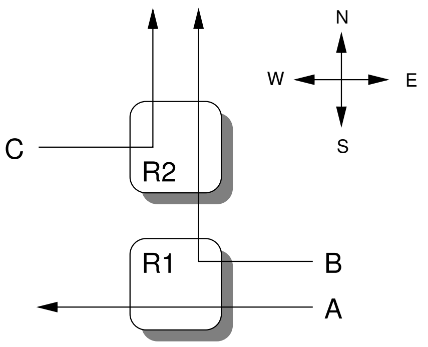


Figure 5.11: Different packet handling orders.

acknowledgement latency is measured. So, each counted acknowledgement latency corresponds to one of the 256 nodes in the MoC(4,2) network.

From the distribution can be seen that the peaks for 4 flitbuffers are shifted slightly towards the right with respect to the peaks for one flitbuffer. This implies that the network latencies of acknowledgements are generally higher in the configuration of 4 flitbuffers than in the single-flitbuffer configuration. Evidently, this contributes to the decrease of throughput for 4 flitbuffers (see Figure 5.9). Also, the distribution shows that the smallest latencies are achieved with 16 flitbuffers, followed by the configuration of 64 flitbuffers. This corresponds with the results of Figure 5.9.

To investigate the reason behind the increased latency of the acknowledgements in the case of 4 flitbuffers, Figure 5.13 gives a more microscopic view of what happens within the network. It shows the differences in the average time that packet headers are blocked within the 16 Clos clusters of the MoC(4,2) for the configuration of 4 flitbuffers compared to the single-flitbuffer configuration. Each bar corresponds to a cluster located in the 4 $\times$ 4 mesh of the MoC(4,2). It might be interesting to know that the hotspots are placed in the four corner clusters.

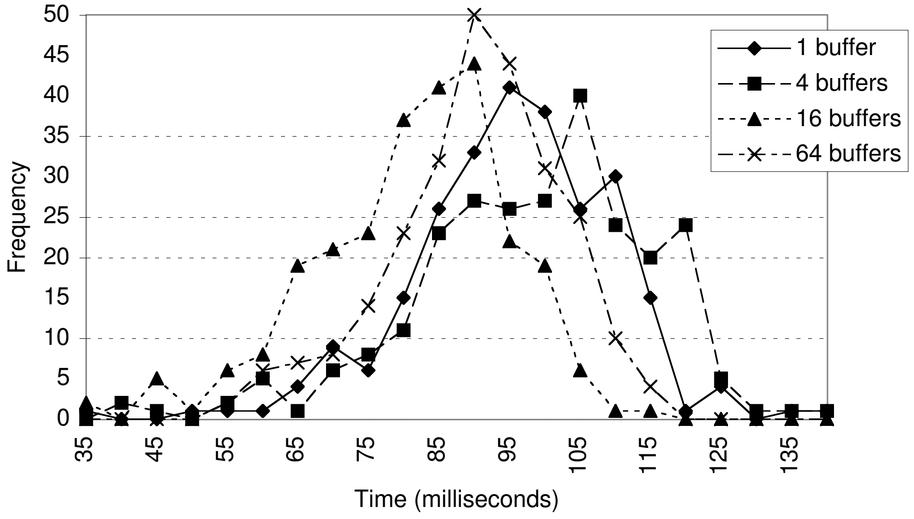


Figure 5.12: Distribution of the cumulative time that nodes waited for acknowledgements in *hotspot* on a MoC(4,2) using messages of 512 bytes.

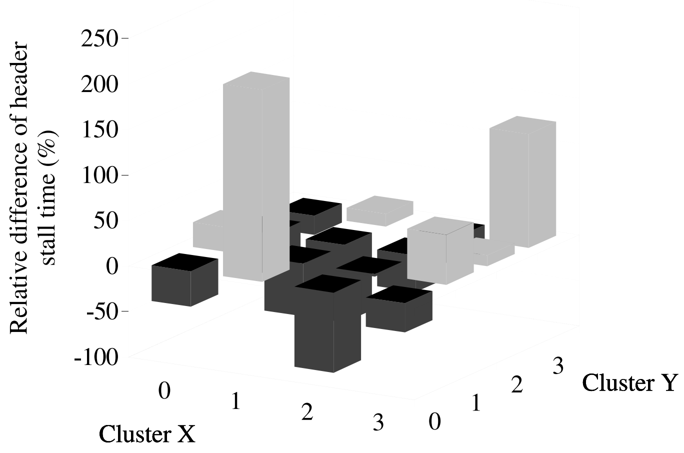


Figure 5.13: The relative difference in average header blocktime within a certain cluster of the MoC(4,2) network. These results are for *hotspot* with messages of 512 bytes and 4 flitbuffers and are relative to the results of the single-flitbuffer configuration.

Although in many clusters a decrease of header stall delay is measured (dark bars), the stall delays in several other clusters have increased considerably (light bars). One cluster in specific (the big peak at the front) is forming a large bottleneck, as we measured an increase of roughly 210% for this cluster. Apparently, the larger number of flitbuffers creates new or amplifies existing hotspots within the network. These hotspots may on their turn affect a large quantity of packets, including those on the critical path of the application (e.g. acknowledgements). This can result in a decrease of overall throughput, as shown in Figures 5.9 and 5.10.

5.4.2 Routing strategies

The non-adaptive routing strategy of the node routers is reasonable simple to implement. Nevertheless, this strategy might not fully exploit the potentials of the layered mesh structure in the Mesh of Clos(4,2) network. Adaptively routing the packets to one of the four layered meshes may result in a better utilisation of network resources. By doing so, the packets within a message can arrive out of order at the destination, as they may travel through distinct mesh-layers with potentially different latencies. Therefore, additional support is required in order to reconstruct a message using the correct sequence of packets.

Several adaptive node router strategies have been simulated to investigate the performance effects on the communication loads. To model the reconstruction of messages, an extra latency has been added at the VCP for each message it receives. The routing within the 2D meshes is still performed deterministically. So, deadlock prevention is not necessary as packets cannot switch from mesh-layer during their journey and thus no "inter-layer" dependency cycles can be formed.

The first two adaptive routing strategies we have examined are *Random* and *RoundRobin*. They route packets unconditionally, which means that they do not examine the state of the output channels. *Random* randomly selects a channel to a layer and *RoundRobin* selects a channel in round robin fashion. Without contention, the selected channel by *RoundRobin*,

Routing strategy	*uniform* (%)	*hotspot* (%)	*hotregion* (%)	*partner* (%)	Overall av.(%)	σ	$\equiv/\uparrow/\downarrow$
Random	0.2	-1.1	-2.1	1.1	-0.5	4.0	$\equiv$
RoundRobin	0.4	-1.7	-0.2	-1.3	-0.7	1.7	$\downarrow$
IdleFixed	4.7	1.5	1.4	1.9	2.4	3.0	$\uparrow$
IdleRandom	2.8	2.6	0.5	3.2	2.3	3.2	$\uparrow$

Table 5.4: Average increase of throughput due to adaptive routing for the different types of communication loads.

which is the least recently used channel, will have the greatest chance of being idle. In the third strategy, called *IdleFixed*, packets are routed along an idle channel. If more idle channels are available, then one is selected at random. Are all channels busy, then this strategy falls back on the original deterministic scheme (see Section 5.2.1). Finally, the *IdleRnd* strategy is similar to *IdleFixed* with the difference that a channel is randomly selected if there are no idle channels available.

Table 5.4 gives an overview of the relative performance effects due to adaptive routing, as compared to the performance of the original non-adaptive routing strategy. It contains the average throughput gain for each communication load, the overall average increase of throughput and the standard deviation of this overall average. Furthermore, the last column indicates whether a routing strategy is statistically equal to ($\equiv$), more effective ($\uparrow$) or less effective ($\downarrow$) than non-adaptive routing when using a confidence interval of 98%. All numbers are percentages and have been averaged over a wide range of message sizes.

The results clearly show that *Random* and *RoundRobin* are the least effective adaptive routing strategies. For most non-uniform communication loads, they actually decrease the throughput. Statistically, the performance of *Random* is not significantly different from that of the non-adaptive strategy. This is caused by the irregular performance behaviour of the *Random* strategy, as illustrated by the large standard deviation that is obtained. So, besides the many throughput decreases, there were measured quite a few increases of throughput.

The *RoundRobin* strategy, on the other hand, cannot compete with the original non-adaptive scheme. Within the given interval of confidence, its communication performance is inferior to that of non-adaptive routing.

IdleRand and *IdleFixed* achieve the highest average throughputs for all communication loads. The overall average performance of both strategies is nearly identical, with *IdleFixed* performing slightly better. Using the confidence level of 98%, it can be concluded that these adaptive routing strategies are more effective than the non-adaptive scheme. Despite this, the throughput improvements are still marginal. Judging from these results, it might not be worthwhile to add extra complexity to the hardware or to the software in order to support adaptive routing at the node routers.

5.5 Discussion

In this chapter, we presented an evaluation study of wormhole-routed Mesh of Clos networks. This type of network, which combines the mesh and Clos topologies, addresses the tradeoff between realisability and efficiency. Simulation experiments with different types of communication loads indicate that the Mesh of Clos networks with Clos structures of height 2 are potentially less prone to contention than flatter mesh-based networks. Under circumstances of congestion, throughputs of up to 75% higher than those for a normal mesh network were measured. Moreover, it was found that the Mesh of Clos network of height 1 generally suffers from high contention. For all applied communication loads, its communication performance is inferior to that of a normal mesh network.

It was shown that adding flitbuffers to router devices does not guarantee a better communication performance. Although many communication loads gain some benefit from a larger number of flitbuffers, the performance impact is not as predictable as one would expect. In several cases, a decrease of throughput was measured when enlarging the buffer space. This is due to the additional network traffic and the altered stall delays of packets that the extra flitbuffers cause. These effects may change the order in which packets are handled by the routers within the network, which subsequently can affect the delays of packets that are on the critical path of the application. This may, in some extreme cases, lead to a degraded overall communication performance.

For a Mesh of Clos network of height 2, which contains four independent layers of two-dimensional meshes, we investigated several strategies to adaptively route packets to these mesh-layers. Of the four adaptive schemes that have been examined, only two obtain a slightly higher throughput compared to a straightforward non-adaptive strategy. These two strategies, which both make routing decisions based on the state of the communication channels, achieve an average throughput increase of roughly 2.4%. So, at first sight it might not be worthwhile to invest in the extra hardware or software necessary for adaptive routing.

Throughout this study, we have assumed that routing within the mesh structures is performed deterministically (see Section 5.2.1). However, applying (partially) adaptive routing within a mesh, combined with a strategy to adaptively route packets to a particular mesh-layer, may result in another gain of throughput. This may therefore be an issue for future research.

Chapter 6

Data prefetching for the TriMedia[†]

"Predicting the future is easy. It's trying to figure out what's going on now that's hard."

Fritz R. S. Dressler

The memory design community has not been able to keep up with the rapid advances in microprocessor technology of the last two decades. As a consequence, a large gap between the performance of microprocessors and main memory developed. To a certain extent, caches are capable of reducing this performance gap. However, as the speed of microprocessors is still improving with great pace, the delays which are due to cache misses (the so-called *cache penalties*) become an increasingly dominant factor in the execution time of programs. In many cases, the cache performance, and thus the average memory latency experienced by the microprocessor, can be improved by simply increasing the cache size or the number of levels of cache. However, there are several application domains for which this solution does not help. In particular, the applications that suffer from a large number of compulsory misses (caused by references that have not been referenced before) are generally not served by increasing the amount of cache space. For example, in multimedia applications, calculations are often applied to one or multiple data (e.g. video) streams. This stream processing usually exhibits no to little re-use of data (i.e. a low temporal locality), which results in a high rate of compulsory cache misses.

Prefetching is a technique that can be used in order to hide the latencies associated with compulsory cache misses. By bringing a data element into the cache (or some other fast memory) before it is actually referenced, the access to the memory is performed simultaneously with computation. The methods for prefetching can be divided into *binding* and *non-binding* types. Binding prefetching brings the data in and assigns it to a specific location, such as a register or a location in memory, from which it can later be used. For example, the *speculation* technique proposed for the new EPIC architecture [51] from Intel and HP can be regarded as binding prefetching. By contrast, non-binding prefetching is only a *hint* to the memory system to try to bring data closer to the microprocessor (e.g. fetch it into the data cache) such that a later binding memory reference will complete much faster. Ideally, all these future memory references will hit in the memory where the prefetched data is

[†]This chapter is based on [112, 113, 134].

stored. This implies that the prefetches should always retrieve the correct data and that this data is always available in time. But, even in the case the data is still being prefetched and is only partly available at the time it is referenced, there usually still is a performance gain with respect to a plain (cache) miss. On the other hand, the danger of non-binding prefetching is *trashing*: it might fetch data which is not going to be used, and which in its turn may replace valuable data from the cache.

This chapter presents an evaluation of several non-binding prefetching techniques. We study traditional methods found in literature and we assess a novel prefetching approach. The motivation behind this work is to find methods for reducing the average memory latency of future versions of the Philips TriMedia microprocessor. The TriMedia, which will be discussed in the next section, is a VLIW processor which is specialised in the processing of multimedia applications [127]. Hence, our focus is on this particular class of applications. Also, we restrict our view to data prefetching only and do not take instruction prefetching into account.

The most important contribution of this study to the Mermaid work is that it illustrates that Mermaid's computational model provides enough architectural detail in order to evaluate uni-processor behaviour. This model is easily modified to allow the evaluation of data prefetching techniques. We demonstrate that the resulting model obtains good performance predictions.

6.1 The Philips TriMedia

The TriMedia is a processor architecture for high-performance processing of multimedia applications in which high-quality video and audio is involved. Figure 6.1 shows the block diagram (taken from [47]) of the first implementation of TriMedia's architectural family, the TM-1. The heart of the TM-1 consists of a general-purpose VLIW processor core which coordinates all on-chip activities. Besides the VLIW core, there are several DMA-driven multimedia I/O units and co-processors. The I/O units format the incoming/outgoing data such that the software which is processing the media can be made efficient. The multimedia co-processors operate in parallel with the VLIW core and perform operations specific to important multimedia algorithms, such as image filtering and resizing and colourspace conversion. This study only deals with the VLIW core of the TriMedia architecture.

The TM-1 VLIW core, of which the block diagram is shown in Figure 6.2, is a 32-bit processor. Because of its VLIW nature, instructions contain five *slots* in which RISC-like operations can be scheduled. At most two of the operation slots can be used for memory accesses (load or store operations). To support predicated execution [56, 87], each instruction slot contains a guard register that determines whether or not the operation should be executed. Moreover, the instructions are stored in a compressed format and are decompressed on-the-fly when they are issued.

Each operation within an instruction is routed to one of the 27 execution units. These execution units read and write from/to a single register file containing 128 32-bit registers. Since an instruction can hold up to five operations, ten read ports (for the operands) and five write ports (for the results) are needed. In addition, five 1-bit read ports are required for reading the bit that is responsible for the guard value.

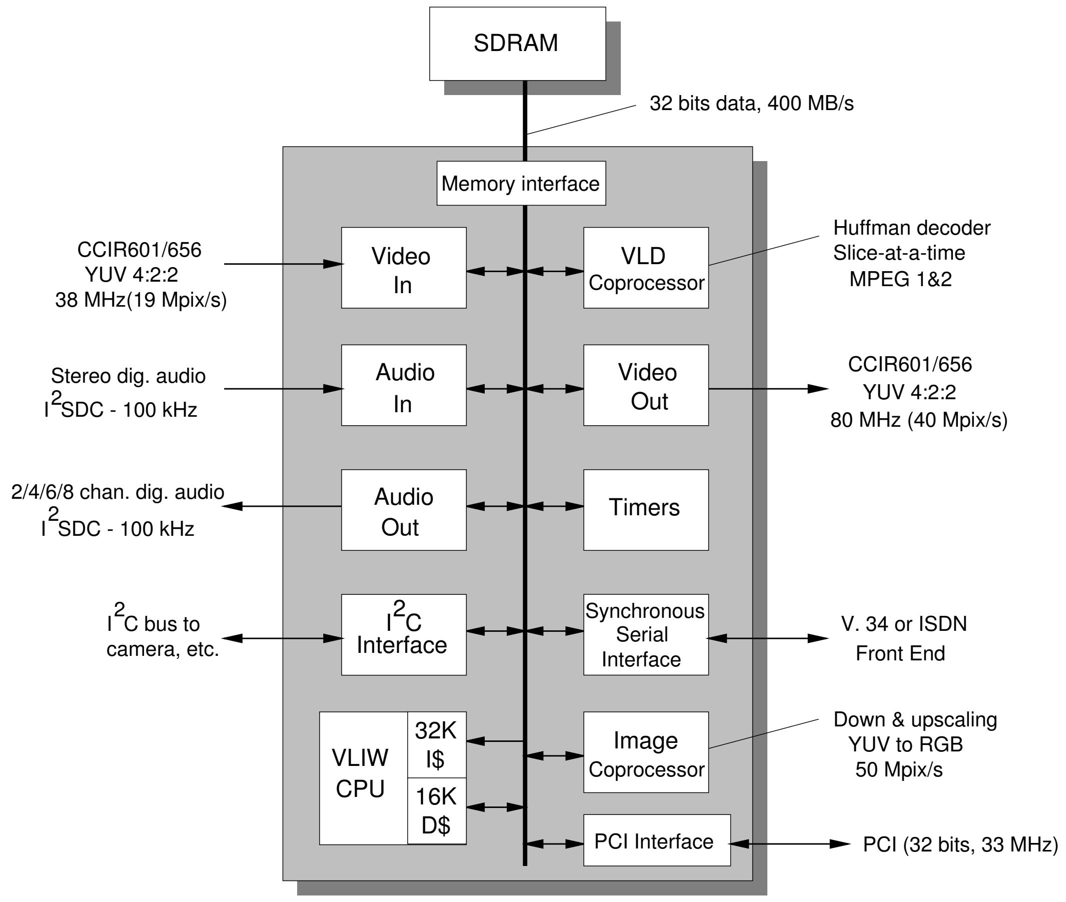


Figure 6.1: The TriMedia TM-1 block diagram.

6.1.1 The TM-1 caches

The TM-1 core contains a 32 Kb instruction cache and a 16 Kb data cache. Both caches are 8-way associative with 64-byte cache blocks and apply some sort of pseudo-LRU, called hierarchical LRU, for replacing the cache blocks. The data cache uses the copyback and write-allocate policies for write operations [52]. Furthermore, the data cache is non-blocking, which implies that it can handle memory transfers and processor requests simultaneously, provided that the processor requests hit in the cache (i.e. memory transfers and processor requests do not interfere).

In order to reduce the stall time of the processor for read operations, the data cache uses a *critical word first* read policy [52] and, in addition, applies so-called *streaming*. In streaming, the cache updates a special scoreboard each time the cache receives a piece of data from the bus. The scoreboard holds the per-word valid bits of the fetched cache block. This may reduce the duration of processor stalls in the case data is required which is already actively being retrieved by the cache. Without streaming, and thus using only a single valid bit for an entire cache block rather than per-word valid bits, the processor is forced to wait until the whole cache block has been retrieved.

As a TM-1 instruction can hold two memory operations, the data cache is dual-ported. This allows the two memory operations to access the cache in parallel, provided that both operations access a different cache bank (the data cache is composed of 8 banks).

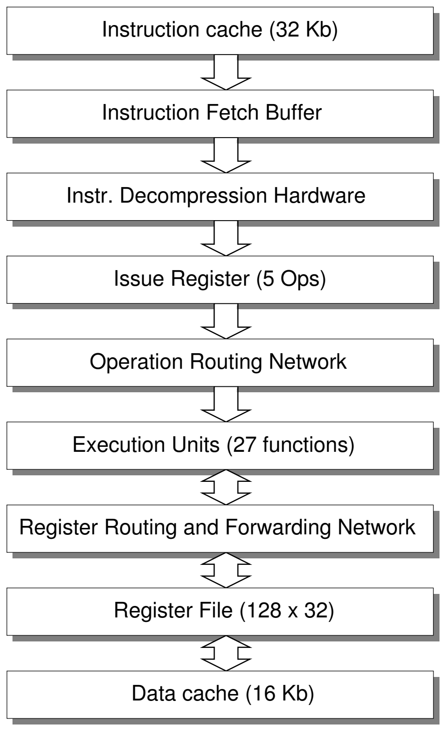

Instruction cache (32 Kb)
Instruction Fetch Buffer
Instr. Decompression Hardware
Issue Register (5 Ops)
Operation Routing Network
Execution Units (27 functions)
Register Routing and Forwarding Network
Register File (128 x 32)
Data cache (16 Kb)

Figure 6.2: The TriMedia TM-1 VLIW core.

6.2 Data prefetching

Prefetching is not a new technique. In 1982, for instance, Alan Jay Smith wrote a well-known paper on caches which discusses a hardware-based prefetching scheme [131]. As a matter of fact, prefetching merely regained the interest of microprocessor architects in the past few years because of the expanding gap between processor and memory performance. As stream-processing applications, such as multimedia applications, have become increasingly popular, it is recognised that prefetching may be a viable technique to reduce the average memory latency.

In this study, we evaluate several (non-binding) techniques to prefetch data streams. Because a wide variety of these so-called *stream prefetching* methods exists, we first discuss the different efforts that have been made in this field according to a classification. This classification identifies the two actions of which stream prefetching is composed:

- *Stream detection*:
  Detects when an application is performing operations on data-streams.

- *Issuing prefetches*:
  Request the prefetch engine (which is usually located in the data cache) to regularly prefetch a certain amount of data. These prefetch requests should be controlled so that no or little trashing occurs.

Both the detection and the issuing can either be performed statically (by the compiler or programmer) or dynamically (in hardware). The different detection/issuing combinations are shown in Table 6.1. In this table, a checkmark means a valid combination, whereas a

		Issuing prefetches	
		Static	Dynamic
Stream detection	Static	√ (Section 6.2.1)	√ (Section 6.2.3)
	Dynamic	×	√ (Section 6.2.2)

Table 6.1: Classification of different types of stream prefetching.

cross denotes a combination which is infeasible to implement. In the following sections, we will discuss each of the valid combinations. For the sake of convenience, we only refer to the compiler when describing static techniques. However, in most cases, these techniques can also be applied by the programmer.

6.2.1 Static detection, static issuing

A straightforward way to prefetch data is to add a *scalar prefetch* instruction to the processor's instruction set. This instruction, which is inserted in the code at strategic places by the compiler, instructs the prefetch engine to prefetch a certain cache block. So, the compiler must detect where prefetch instructions have to be inserted and, additionally, it should insert enough prefetch instructions to prefetch the entire stream in time. This software-driven type of prefetching is becoming increasingly popular. For example, the HP PA-8000 features a scalar prefetch, which turns out to be quite effective for some applications [122].

Several methods have been proposed to perform the insertion of prefetch instructions at compile-time. A well-known example is the compiler algorithm devised by Mowry et al. [95]. This algorithm inserts prefetch instructions into code that operates on dense matrices by identifying the references that are likely to be cache misses. An interesting alternative has been proposed by Zucker et al. in [147]. They apply a compile-profile-compile cycle to insert the prefetch instructions at the appropriate places. During the profile phase, a stream detection technique derived from hardware prefetching (which will be discussed in the next section) is used to identify the data streams within the program.

The major disadvantage of scalar prefetching is the increase of the number of instructions. For each cache block that has to be prefetched, at least one instruction has to be executed by the processor core. Additionally, the prefetch instructions also affect the compiler optimisations. To reduce the number of prefetch instructions, for instance, loop unrolling and loop splitting is often required [95]. These techniques are employed in order to prevent the insertion of superfluous prefetch requests referring to memory locations that are already in the cache. As a consequence, the number of prefetch instructions is limited to the ones that are really needed, i.e. the ones with a prefetch address that crosses a new cache block boundary. However, such optimisations may be hard to perform when multiple streams are involved which are misaligned relative to each other.

The compiler also has to take care that prefetching is started in time. This requires, for instance, the addition of so-called *prefetch prologues* to establish the appropriate *prefetch distance* before the stream is actually accessed. This prefetch distance is the distance that the prefetches "run ahead" with regard to the actual program references. An example of scalar prefetching is shown in Figure 6.3b.

```
for (i = 0; i < N; i++)
    sum = a[i] + sum;
```

(**a**)

```
for (i = 0; i < 3; i++)
    prefetch(&a[i]);
for (i = 0; i < N - 3; i++) {
    prefetch(&a[i+3]);
    sum = a[i] + sum;
}
for (; i < N; i++)
    sum = a[i] + sum;
```

(**b**)

Figure 6.3: Statically detected, statically issued prefetching. Code fragment (**a**) shows the original loop. The code in (**b**) is augmented with scalar prefetches. The new loop in front of the original loop is the prefetch prologue establishing a prefetch distance of 3 prefetches.

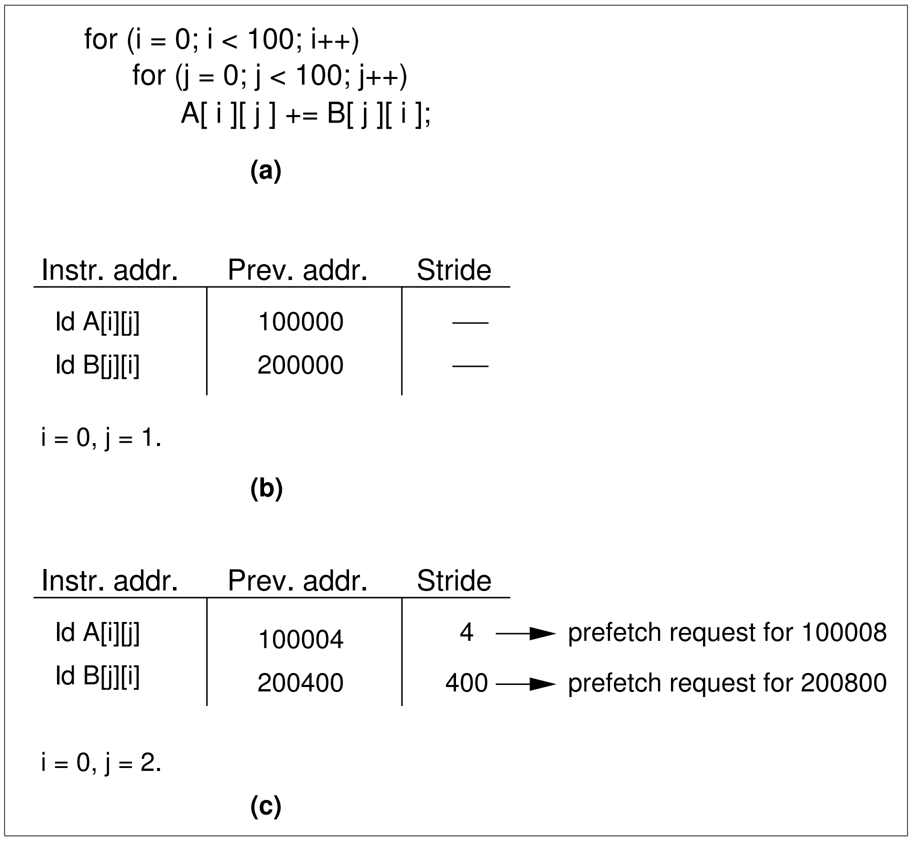
6.2.2 Dynamic detection, dynamic issuing

Opposed to software (scalar) prefetching is pure hardware prefetching. In this technique, both the detection of the streams and the issuing of prefetch requests are performed at run-time by a hardware prefetch engine. Note that when the streams are detected dynamically by means of hardware, the prefetch requests must be issued by hardware.

Smith was the first to propose a hardware-based prefetching method [131]. In his one-block-lookahead (OBL) scheme, the cache prefetches cache block $i+1$ whenever a demand miss brings block i into the cache. Jouppi expanded this idea with his proposal for stream buffers [64]. In this scheme, a miss which causes block i to be brought into the cache triggers prefetch requests for the blocks $i + 1$ to $i + n$. Subsequently, the prefetched blocks are stored in a special stream buffer. As there is often more than one stream active at the same time, Jouppi also proposed a multi-way stream buffer which allows for maintaining multiple streams. To reduce the amount of trashing in the stream buffer, Palacharla et al. devised a filtering technique which detects the regular access pattern of a stream [103].

Another hardware prefetching method, proposed by Fu and Patel [42], introduces a hardware table which records the history of memory references to identify streams and to predict future references. This table, which is called the Stride Prediction Table (SPT), stores the instruction address of memory references together with the data address that is referenced. At a new memory reference, the program counter (PC) is searched for in the SPT. If the PC is found in the SPT, a *stride* can be calculated by subtracting the data address stored in the SPT-entry from the data address of the current reference. So, the stride equals the distance between two consecutive memory references (made by the program) in a stream. Subsequently, a request is issued to prefetch data from the location which is anticipated to be accessed next, being the current data address plus the stride. This type of hardware prefetching is illustrated in Figure 6.4. Hence, in this method, the issuing of the prefetches is *synchronised* using the instruction addresses of memory operations.

Chen and Baer proposed several optimisations to this scheme [22], of which we discuss the two most important. In order to reduce the number of erroneous prefetches, they added



```
for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
        A[ i ][ j ] += B[ j ][ i ];
```

(a)

Instr. addr.	Prev. addr.	Stride
ld A[i][j]	100000	—
ld B[j][i]	200000	—

i = 0, j = 1.

(b)

Instr. addr.	Prev. addr.	Stride
ld A[i][j]	100004	4 $\longrightarrow$ prefetch request for 100008
ld B[j][i]	200400	400 $\longrightarrow$ prefetch request for 200800

i = 0, j = 2.

(c)

Figure 6.4: Hardware prefetching using a stride prediction table (SPT). In figure (**a**), a code fragment is shown which operates on two data streams A and B. Situation (**b**) shows the state of the SPT after the first iteration of the inner-loop. The instruction addresses of both load operations have been inserted in the SPT together with the data addresses which have been loaded. In figure (**c**), the situation is sketched after the second iteration of the inner-loop. Both load operations did hit in the SPT after which a stride has been calculated from the new data addresses and the SPT-stored previous addresses. Subsequently, the SPT-stored data addresses are updated with the new addresses and there are two prefetch requests issued.

state information to each SPT entry. This state indicates whether or not a prefetch request should be issued at an SPT hit. For example, the state of an SPT-entry is valid (implying it can issue prefetch requests) only when a constant stride is measured within the stream. By doing so, irregular access patterns do not cause erroneous prefetches as the SPT just ignores these accesses. Chen and Baer also describe a technique to use a so-called lookahead program counter (LA-PC) instead of the normal PC to search the SPT. With the help of the processor's branch prediction table, the LA-PC runs ahead with respect to the PC and is therefore able to increase the prefetch distance.

Most of these SPT-based methods bring the prefetched data into the data cache. There is, however, also work performed on SPT-based prefetching to a special stream cache which is similar to the previously mentioned multi-way stream buffer from Jouppi. This stream cache may act as a L2 cache or it may be queried in parallel with the L1 data cache [146].

The fact that SPT-based prefetching relies on the instruction addresses of memory operations for the identification of streams and synchronisation of prefetches has several consequences. First, the SPT should be reasonably large as every memory operation might indicate the start of a stream. Second, the program counter has to be routed to the SPT logic (which might be located in the data cache), thereby affecting the processor's logic. Hence, this method does not allow to simply exchange the normal data cache for a prefetching data cache. A potential solution to this problem is to perform both the stream detection and prefetch synchronisation solely on the basis of data addresses. Although this is quite complex, attempts are currently being made to support this type of prefetching [62]. Third, loop unrolling may affect the performance of SPT-based prefetching as one stream is split into multiple smaller streams which all have to fit in the SPT. Other drawbacks of SPT-based prefetching are the delay before a stream is actually detected (it takes at least two memory references) and the fact that the SPT may issue erroneous prefetch requests (e.g. it issues a request for the cache block beyond the end of a stream).

6.2.3 Static detection, dynamic issuing

Prefetching can also be done in a hybrid manner, thus by both software and hardware. Doing so, the best of both worlds can be combined. For example, the problem of the large number of extra instructions as experienced in pure software prefetching is solved by issuing the prefetch requests dynamically rather than statically. Moreover, by statically detecting streams, there is more control over the prefetching than in case of dynamic detection. As a consequence, the number of erroneously prefetched cache blocks can be minimised or even reduced to zero. Additionally, the amount of required hardware resources for hybrid prefetching is smaller than for SPT-based prefetching. This because the latter form of prefetching also records information on memory references that *may* behave as a stream.

In hybrid prefetching, the compiler detects streams within a program and inserts special *stream prefetch* instructions at these places. The stream prefetch instructions command a special piece of hardware to issue a prefetch request from time to time. The term "from time to time" is, however, rather subtle as prefetches should be started way before their data is actually needed. So, like pure hardware prefetching, there should be some synchronisation between the actual references of the program and the issuing of prefetch requests by the hardware. Basically, we can distinguish three types of synchronisation for hybrid prefetching: using a time interval, PC-based synchronisation and synchronisation on data addresses. The latter one is a method we have recently proposed [134].

Time interval

First, one can use a *time interval* for synchronisation. This means that the stream prefetch instruction should specify a time interval in cycles, allowing the hardware to set some timer in order to repetitively issue prefetch requests. Clearly, this method puts high demands on the compiler to figure out when exactly certain data elements should be prefetched. Especially in the case of conditional branches, it may be hard to keep this time-based prefetching in sync with the actual application references. Because of these disadvantages, this technique is hardly ever used. Therefore, we will not consider it any further.

PC-based synchronisation

In *PC-synchronised prefetching*, the program counter is used to synchronise prefetches with the actual references. This method is identical to the synchronisation in SPT-based prefetching. The stream prefetch instruction initialises an entry in a special hardware table, called the Prefetch Information Table (PIT), to drive the prefetch engine. The instruction provides the PIT with information on the instruction address which should trigger a prefetch (i.e. on which the prefetches are synchronised), the data address at which should be prefetched, the stride with which should be prefetched and a count specifying the number of prefetches that should be performed. In addition, the preferred prefetch distance, or *runahead*, can also be specified. This runahead is established by directly issuing the appropriate number of prefetch requests at the time the PIT entry is initialised. Naturally, there are multiple entries in the PIT, allowing multiple stream prefetches to be active at the same time. At each memory access of the program, the PIT checks its entries and may issue a prefetch request accordingly. The latter occurs when the program counter (PC) of a memory access hits in the PIT. After such a PIT hit, and thus after issuing a prefetch request, the information in the relevant PIT entry (e.g. the prefetch address) is updated. In [21], Chen describes the *Hare* prefetch engine which is based on this principle.

This prefetch technique still has several drawbacks. The instruction address specified in the stream prefetch instructions (for synchronisation purposes) may be hard to determine when the prefetch instructions are inserted manually at the source level. Furthermore, like in SPT-based prefetching, the program counter requires to be routed to the PIT. As a result, PC-synchronised hybrid prefetching is not transparent to the processor. Finally, PC-based synchronisation in general may easily be affected by compiler optimisations such as loop unrolling. For example, when unrolling a loop with several streams, a multiple of smaller streams is created which may all have to fit in the PIT (or the SPT, for that matter). To overcome these drawbacks, we have proposed a new hybrid prefetching scheme which is based on data address synchronisation rather than on PC-based synchronisation [134].

Data address synchronisation, a novel approach

Synchronisation on data addresses is similar to PC-based synchronisation as it also uses a PIT. Again, a stream prefetch instruction initialises a PIT entry with information on the data address at which should be prefetched, the stride, the number of prefetches and the runahead. But, rather than specifying an instruction address that synchronises the prefetches, the starting data address of the stream is specified for the synchronisation. This is illustrated in Figure 6.5b. Thus, instead of matching the PC, the data address of memory references is used to determine whether or not there is a PIT hit. At every hit, a new prefetch request is issued after which the synchronisation and prefetch addresses of the PIT entry are updated (using the stride).

Compared to PC-synchronised prefetching, our data address synchronisation scheme has three advantages. First, the program counter does not need to be forwarded to the PIT. Second, data address synchronisation may lead to a significantly smaller PIT because this technique is not affected by compiler optimisations. Unrolling a loop with a stream, for instance, does not result in a multiple of smaller streams in the PIT; there is still one PIT entry for the unrolled stream. Third, data synchronised prefetching is more robust than its PC-

```
for (i = 0; i < N; i++)
    sum = a[i] + sum;
```

(**a**)

```
prefetch(&a[0], N, 4, 3);
for (i = 0; i < N; i++)
    sum = a[i] + sum;
```

(**b**)

Figure 6.5: Statically detected, dynamically issued prefetching. Code fragment (**a**) shows the original loop. In code fragment (**b**), the code is augmented with a data-synchronised stream prefetch instruction. Its parameters include the number of prefetches, the stride in bytes and the preferred prefetch distance (3 in this case).

synchronised counterpart. More specifically, address synchronisation determines *where* the processor is accessing a stream, rather than determining *that* the processor is accessing a stream like PC-based synchronisation does. This allows, for example, prefetching with a stride that is different from the one used for the actual referencing of the stream. Especially in the presence of conditional data accesses or when the stride is regular but not always fixed within a stream, this may improve the ability to prefetch considerably. For example, if a stream is accessed using two strides a and b in an alternating manner, then the stream can be prefetched by synchronising on the data addresses referenced by the strides a and b separately.

On the other hand, as there is no such thing as a free lunch, there is one drawback of data synchronised prefetching. The synchronisation addresses which are searched for in the PIT change dynamically over time (at each PIT hit, the synchronisation address is updated using the stride). This implies that the PIT must be implemented using a fully associative memory. By contrast, PC-synchronised prefetching can use a table which is not fully associative (e.g. set-associative) because the PIT is indexed by static instruction addresses. However, since the PIT can be kept reasonably small when using data synchronised prefetching, its fully associative implementation should not be a real problem.

6.3 The simulation methodology

For this performance evaluation, we have used Mermaid's single-node computational model as a starting framework for the simulator. As we are interested in TriMedia's memory hierarchy, the operation-traces driving the simulator only consist of memory-related operations. This includes instruction fetches, load and store operations and, when required, special prefetch operations which we added to simulator. The traces were obtained by executing and tracing applications in TMsim [124], which is a cycle-accurate simulator of the TriMedia processor. The resulting output trace was then converted to our operation format. To reduce the size of the operation-traces, we applied both Lempel-Ziv compression and relative tracing using page addresses (see Section 2.1.2).

With respect to the architecture model, we preserved the basic infrastructure as shown in Figure 3.13a. Only the functionality of several components has been changed. The processor component, for example, now acts like a pseudo 1-CPI processor. At every instruction-fetch operation, which represents the issuing of a TriMedia instruction comprising of five operation-slots, it accumulates the cycle count to model the execution of the instruction. No instruction cache is modelled as its hit-ratio is assumed to be perfect. After an instruction-fetch, there may follow one or two memory-reference operations (as the TriMedia instruction allows two memory I/O operations per instruction) which are explicitly simulated by the data cache component. At a data cache hit, no extra latency is counted. In this case, the memory operation is just performed in one cycle. Thus, here we assume that in case of a load operation, the cache access latency is covered up by scheduling the load early enough before the data is actually needed. Hence, the data cache component only models penalties due to resource conflicts, cache misses or to synchronisations. Therefore, the real (i.e. measured) CPI can be calculated as follows:

$$\text{CPI} = \frac{(I_{read+write} \times \text{Average\_DataCache\_Penalty}) + I_{total}}{I_{total}}$$

where I_{total} denotes the total number of instructions and $I_{read+write}$ is the number of memory references.

The processor component also allows for filtering the incoming operation-trace to represent different execution behaviour. For example, a SIMD-like trace can be obtained by combining several arithmetic instruction-fetch operations into one event. Here, an arithmetic instruction is defined by an instruction-fetch operation which is not followed by a memory-I/O operation but immediately by the next instruction-fetch operation.

The data cache component models the non-blocking TM-1 data cache. In addition, it also provides support for software-based, hardware-based and hybrid data prefetching. This component is discussed more elaborately in the next section. Finally, the bus and memory components are less sophisticated as they just are simple delay mechanisms accounting for the access and transfer delays of the 64-bit bus and the SDRAM memory.

6.3.1 The data cache model

The data cache component of the simulator models the geometry and functionality of the TM-1 data cache as it was described in Section 6.1.1. In addition, the cache model also supports an allocate-but-not-fetch-on-write policy, also referred to as the *write-validate* policy. For this purpose, a valid bit is kept for each separate byte within a cache block. We have included this policy because of the fact that stream-processing applications rarely read their results after writing them to memory. So, this implies that it is unnecessary to bring the data into the cache after it has been written.

Furthermore, to allow data prefetching, the data cache model features a Prefetch Queue (PQ). In this PQ, the issued prefetch requests (originating from either scalar prefetch instructions or from the SPT/PIT) wait to be consumed by the data cache. The PQ is a simple FIFO buffer which ignores new prefetch requests in the case it is full. Moreover, the data cache model includes an SPT for hardware prefetching and a PIT for hybrid prefetching. Both tables apply an LRU replacement strategy. By default, the SPT is 4-way set-associative containing 32 sets and the PIT is 16-way fully associative. The SPT uses state bits, similar to ones proposed by Chen and Baer [22], to guarantee that prefetch requests are only issued when a stream exhibits a constant stride.

To avoid the situation in which the PQ is flooded with prefetch requests for the same cache block, the SPT only issues a request when the prefetch address points to a "new" (not yet requested) cache block. For example, when the address of a prefetch request crosses a cache block boundary, the request is approved and issued. So, the SPT will never issue two prefetch requests for the same cache block to the PQ. This approach is not directly applicable for hybrid prefetching as the PIT does not store the previously referenced data address. Hence, for this type of prefetching we have taken other measures to limit the PQ flooding. First, the PIT checks the contents of the cache and only issues a prefetch request when the required cache block is not present (this is also done by the SPT). Second, in data-synchronised hybrid prefetching only one prefetch request per cache block is issued by manipulating the stride in the calculation of the prefetch address. More specifically, the PIT checks the given stride (specified in the stream prefetch instruction) and if this stride is smaller than the cache block size, then the PIT uses the cache block size (instead of the stride) to calculate the next prefetch address. For PC-synchronised hybrid prefetching, this stride manipulation is not straightforward as this would make the synchronisation process more complex (it may, for instance, require an extra counter determining when to issue a prefetch request at a PIT hit). Therefore, we did not apply this technique to PC-synchronised hybrid prefetching, implying that the PIT may issue multiple prefetch requests for the same cache block. For this reason, this prefetching method might require a larger PQ.

We have also addressed the potential weakness of the traditional SPT-based prefetching method associated with the timing of prefetches, that is, the prefetch requests are issued only one iteration before the data is really needed. If the loop body is too small, the prefetched data may arrive too late for the next access. Unlike Chen and Baer, who introduce a lookahead-PC [22] to solve this problem, we have chosen for a simpler approach. Our technique, which is called *early-prefetch*, exploits the situations in which the stride is smaller than the cache block size. Whereas normal SPT-based prefetching ignores a prefetch request that still refers to the same cache block as the previous request, early-prefetch simply issues a prefetch request for the succeeding cache block in that case. Because this allows multiple prefetch requests (for the next cache block) to be issued from different accesses within one cache block, the SPT prevents this from happening by explicitly checking whether or not an early-prefetch request has already been made or not. To illustrate the early-prefetch technique, consider Figure 6.6. In this figure, the arrows indicate the stream memory-references made by the program. Furthermore, the blocks refer to the cache blocks which are being accessed. When normal SPT-based prefetching is applied, the prefetch requests are issued one iteration before a next cache block is accessed, as indicated by the arrows tagged with "normal" in Figure 6.6. However, in the early-prefetch mode, prefetching starts earlier. If, at the first access in a "new" cache block, the next access will still be located in the same block, then this access would normally have been ignored. But in the case of early-prefetch, a prefetch request for the succeeding block is already being issued. This is indicated by the arrows tagged with "early pf" in Figure 6.6. Doing so, the early-prefetch technique may increase the prefetch distance significantly. On the other hand, for streams that are small enough to fit into one cache block, this technique may increase the number of erroneous prefetches (as it will prefetch the next cache block). Moreover, early-prefetching is only beneficial when the data streams are contiguously laid out over the cache blocks. If the stride within a stream is equal to or larger than the cache block size, then early-prefetching has no effect, implying that the prefetch distance remains unchanged.

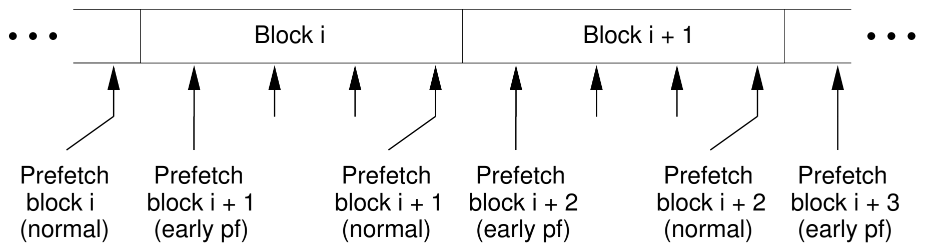

• • •
Block i
Block i + 1
• • •
Prefetch block i (normal)
Prefetch block i + 1 (early pf)
Prefetch block i + 1 (normal)
Prefetch block i + 2 (early pf)
Prefetch block i + 2 (normal)
Prefetch block i + 3 (early pf)

Figure 6.6: The difference between normal SPT-based prefetching and prefetching using the early-prefetch mode.

As all prefetching is done into the data cache, this may lead to the removal of valuable data, i.e. trashing. To reduce the effect of trashing, the cache model allows to specify a subset of cache blocks which may be the target for prefetched data. One could, for instance, configure only two of the 8 blocks in a cache set for prefetching. In that case, the remaining six blocks within the set are strictly used for normal data references. Normal data references can, however, use the prefetch lines as well. For the prefetch lines, an additional replacement strategy is used, which currently is LRU. Prefetches update both the global strategy (i.e. hierarchical LRU) and the "local" prefetch strategy. As a consequence, normal data references will try to avoid prefetch lines when a lot of prefetching is done, thereby almost guaranteeing exclusive access to the prefetch lines by prefetches only. On the other hand, if there is not much prefetch activity, the cache is free to use all the lines within a set for normal data references.

6.4 Experiments

To evaluate the different prefetching techniques for future TriMedia VLIW processors, we have performed a simulation study using a "de-interlacing" application, called Median. This program processes the odd and even frames of an interlaced video stream in order to produce non-interlaced frames. The following pseudo-code shows Median's computational kernel for the processing of one frame (either even or odd):

```
for ( line = 0; line < NR_OF_LINES; line++ )
   for ( pixel = 0; pixel < NR_OF_PIXELS; pixel++ ) {
      process pixel;
   }
```

So, the inner-loop iterates over the pixels in a scan line and the outer-loop over the scan lines within the frame. Unfortunately, this is the only application we were able to study. Access to more advanced programs, such as MPEG-2, was not available. This is because these applications are still in the process of being implemented (or tuned) for the new TriMedia architecture, i.e. the successor of the TM-1. Because our investigation is limited to a single, rather straightforward benchmark, the main purpose of this study is to gain insight into the *relative* behaviour of several different prefetching techniques. By no means, we try to predict absolute performance improvements by extrapolating the results from the Median benchmark.

The evaluation is divided into two studies. In the first study, the performance gain of pure hardware prefetching is compared to that of hybrid prefetching, while the second study investigates the differences between hybrid and scalar prefetching.

6.4.1 Hardware versus hybrid prefetching

In this section, we use two performance metrics: the hit-ratio and the *Memory CPI* (MCPI). The latter specifies the extra fraction of CPI added by cache penalties, or formally:

$$\text{MCPI} = \frac{(I_{read+write} \times \text{Average\_DataCache\_Penalty})}{I_{total}}$$

where $I_{read+write}$ again refers to the number of memory references and I_{total} to the total number of instructions. As was mentioned earlier, the average data cache penalty is due to resource conflicts, cache misses and synchronisations. So, since the simulator is based on a 1-CPI model, the actual CPI equals to *1 + MCPI*. We simulated the address traces of two instances of Median: *SMedian* de-interlaces a 400×20 image (32Kb in total), whereas *LMedian* uses a larger 200×140 image (160Kb). In Table 6.2, the cache hit-ratio and MCPI for both benchmarks are shown when there is no prefetching performed. The table gives the results for both the fetch-on-write and write-validate policies.

SMedian				LMedian			
Write-validate		Fetch-on-write		Write-validate		Fetch-on-write	
Hit-ratio	MCPI	Hit-ratio	MCPI	Hit-ratio	MCPI	Hit-ratio	MCPI
97.37%	0.234	97.43%	0.278	98.32%	0.157	98.34%	0.185

Table 6.2: Hit-ratio and MCPI for a non-prefetching cache with fetch-on-write or write-validate.

The results from Table 6.2 indicate that Median benefits from a write-validate data cache. As the hit-ratios obtained by the fetch-on-write policy are only marginally higher than the write-validate hit-ratios, this suggests that there is not much data which is read after it has been written. So, by not fetching cache blocks at write misses (i.e. the write-validate policy) valuable memory bandwidth is saved. As a consequence, the MCPI values for the write-validate policy are roughly 15% lower than the ones obtained by a fetch-on-write cache. Our goal is to reduce these MCPI values even further using prefetching.

For hybrid prefetching, the kernel of Median (as shown in the previous fragment of pseudo-code) has been instrumented with stream prefetch instructions. Hence, prefetching only occurs during the execution of this kernel. By contrast, with hardware prefetching, prefetch requests can be issued during the execution of the *whole* program. As the Median benchmark also includes several input and output system calls for the video streams, these code segments may benefit from hardware prefetching as well.

Hardware prefetching

For pure hardware prefetching, a 4-way set-associative SPT is used which may range in size from 8 to 256 entries. By default, this SPT operates in normal mode, thus without the early-prefetch optimisation. Unless stated otherwise, the data cache features a single-entry Prefetch Queue (PQ) and contains two prefetch blocks per cache set in order to reduce trashing.

As TriMedia instructions can hold two memory references, which both will use the same instruction address for prefetch synchronisation, the SPT should handle both memory references separately. Given this background, we have studied three possible SPT implementations. First, the SPT can be split into two banks, one for every memory-IO slot in the TriMedia instruction. Indexing such an SPT is done like

$$\text{SPT[instr\_slot][instr\_adr } \textit{mod} \text{ BANKSIZE]}$$

In this case, a memory reference hits the SPT when the entry is valid and the instruction addresses match. We refer to this as a *split* SPT. A second implementation can be realised with one bank where the indexing is done like

$$\text{SPT[(instr\_adr + instr\_slot) } \textit{mod} \text{ BANKSIZE]}$$

Here, the *instr_slot* refers to the memory-IO slot of the TriMedia instruction and assumes either 0 or 1. The addition of *instr_slot* prevents aliasing between two memory operations within one instruction. Like in the previous configuration, a memory reference hits the SPT when the entry is valid and the instruction addresses match. We call this a *shifted* SPT. Finally, a single-banked SPT could use the following index function:

$$\text{SPT[instr\_adr } \textit{mod} \text{ BANKSIZE]}$$

In this case, the SPT is supposed to store slot-identifiers as well. Subsequently, a memory reference hits the SPT when the entry is valid and both the instruction addresses *and* the slot-identifiers match. We refer to this as a *slotted* SPT.

Table 6.3 shows the SPT hit-ratios for the three different SPT implementations: shifted, slotted and split. These results were obtained using a write-validate policy. Clearly, the shifted and split SPTs outperform the slotted SPT for a small number of entries. This can be explained by the fact that the compiler may schedule two stream references into one

	SMedian			LMedian		
SPT size	Shifted	Slotted	Split	Shifted	Slotted	Split
8	85.97%	66.65%	85.87%	88.06%	72.50%	88.03%
16	86.39%	67.74%	87.21%	88.41%	74.06%	89.76%
32	91.15%	70.92%	90.49%	94.41%	77.72%	94.46%
64	92.77%	93.24%	93.31%	98.26%	98.25%	98.23%
128	95.65%	95.23%	95.44%	98.93%	98.74%	98.82%
256	96.77%	96.53%	96.72%	99.23%	99.15%	99.22%

Table 6.3: SPT hit-ratio for the three SPT configurations (using a write-validate policy).

TriMedia instruction. In a slotted SPT, such stream references map to the same SPT set which might cause trashing when there is only a small number of SPT sets available. But, as can be seen from Table 6.3, the differences between all three SPT implementations become marginal when increasing their size.

In Table 6.4, the hit-ratios of a shifted SPT are shown for both the fetch-on-write and write-validate policies. Note that in the case of fetch-on-write there will be prefetches on both read and write operations, whereas in write-validate there are only prefetches on read operations. As SPT hit-ratios are no direct indicator of real performance, the table also shows the obtained MCPI values for the same experiment. The results clearly illustrate that prefetching has significantly decreased the MCPI values as compared to the values listed in Table 6.2. The MCPI reductions we measured are between 16% and 32%. Not surprisingly, the best improvements are for the write-validate policy. In the fetch-on-write policy, a lot of prefetches are, when compared to the write-validate policy, not effective as they are used for write operations only and not for read accesses. These prefetches did, however, prevent other (valuable) prefetches from being executed.

Furthermore, Table 6.4 also indicates that the SPT hit-ratios of write-validate are better than the ones of fetch-on-write for smaller SPT sizes (smaller than 64 entries). This is probably due to several write operations which are not part of a stream but still trash the SPT. Another observation is that improving the SPT hit-ratio (by increasing the SPT's size) does not result in a performance gain (i.e. a lower MCPI). This can be explained by the fact that the number of prefetches issued by both small and large SPTs are identical (these numbers are not shown). The extra hit-ratio of large SPTs does not cause the issuing of many extra (effective) prefetches.

The above conclusions can also be drawn when using a slotted or split SPT rather than a shifted SPT (we do show these results for the sake of brevity). The obtained MCPI values are for all three SPT implementations identical, except when using a small slotted SPT. In the case of the latter, a lower MCPI was measured due to the poor SPT hit-ratio (see Table 6.3).

Table 6.5 shows the cache hit-ratio and the MCPI when the number of prefetch blocks per set in the cache are varied. Limiting the number of prefetch blocks can reduce the amount

SPT size	SMedian				LMedian			
	Write-validate		Fetch-on-write		Write-validate		Fetch-on-write	
	Hit-ratio	MCPI	Hit-ratio	MCPI	Hit-ratio	MCPI	Hit-ratio	MCPI
8	85.97%	0.159	76.36%	0.214	88.06%	0.124	80.99%	0.156
16	86.39%	0.159	86.66%	0.212	88.41%	0.124	89.27%	0.156
32	91.15%	0.159	88.46%	0.212	94.41%	0.124	91.74%	0.156
64	92.77%	0.159	91.69%	0.212	98.26%	0.124	95.84%	0.156
128	95.65%	0.159	94.23%	0.212	98.93%	0.124	97.43%	0.156
256	96.77%	0.159	95.88%	0.212	99.23%	0.124	98.98%	0.156

Table 6.4: SPT hit-ratio and MCPI of a (shifted) SPT with a fetch-on-write or write-validate cache.

Nr. of prefetch lines	SMedian				LMedian			
	Write-validate		Fetch-on-write		Write-validate		Fetch-on-write	
	Hit-ratio	MCPI	Hit-ratio	MCPI	Hit-ratio	MCPI	Hit-ratio	MCPI
1	98.50%	0.164	98.27%	0.416	99.07%	0.126	98.78%	0.324
2	98.53%	0.159	99.04%	0.212	99.08%	0.124	99.40%	0.155
3	98.44%	0.162	99.02%	0.201	99.05%	0.125	99.40%	0.155
4	98.38%	0.164	99.00%	0.214	99.01%	0.122	99.39%	0.155
5	98.36%	0.164	98.99%	0.207	98.98%	0.120	99.39%	0.155
6	98.32%	0.169	98.97%	0.218	98.96%	0.119	99.38%	0.156
7	98.27%	0.169	98.93%	0.216	98.93%	0.121	99.37%	0.158
8	98.25%	0.171	98.88%	0.221	98.91%	0.122	99.34%	0.155

Table 6.5: The performance of different prefetch configurations with fetch-on-write or write-validate.

of cache trashing (e.g. replacing persistent data from the cache) due to prefetching. In this experiment, a 128-entry shifted SPT is used. The results show that one prefetch block is too few when applying the fetch-on-write policy. For the other configurations, the performance differences are marginal and there cannot be detected a clear trend. We have reached the same conclusion when using different SPT types and sizes. It is therefore hard to determine the optimal number of prefetch lines. This suggests that Median does not use a lot of other, more persistent, data besides its video streams. So, the Median program is not a very good measure for this experiment as the outcome might be significantly different for other benchmark programs.

For a write-validate cache and a 128-entry shifted SPT, Table 6.6 shows the results when varying the Prefetch Queue (PQ) size. It shows both the number of cancelled prefetch requests due to a full PQ and the performance impact (hit-ratio and MCPI). Table 6.6 suggests that increasing the PQ size, which results in less cancellations of prefetch requests, only marginally improves the performance. This can be explained by the fact that although the larger PQ size avoids a large number of the cancellations, most of the "saved" prefetch requests are handled too late. This means that the required cache blocks have already been referenced at the time the prefetch requests are handled, after which the requests are dis-

# PQ entries	SMedian			LMedian		
	# Cancel.	Hit-ratio	MCPI	# Cancel.	Hit-ratio	MCPI
1	1438	98.53%	0.159	4044	99.08%	0.124
2	644	98.53%	0.158	3140	99.08%	0.123
4	182	98.54%	0.157	1143	99.09%	0.123
8	65	98.54%	0.156	8	99.13%	0.121

Table 6.6: Performance impact of a larger PQ when using write-validate and a 128-entry shifted SPT.

	SMedian			LMedian		
	Overall hit-ratio	Hit-ratio in kernel	Overall MCPI	Overall hit-ratio	Hit-ratio in kernel	Overall MCPI
No prefetching	97.37%	95.80%	0.234	98.32%	95.80%	0.157
HW prefetching	97.54%	97.54%	0.219	98.44%	97.31%	0.146

Table 6.7: Performance of a data cache with and without prefetching in the inner loop (kernel) of Median (using write-validate and a 128-entry SPT).

carded after all.

Thus far, the prefetch results are obtained by a model in which prefetching is turned on during the whole trace. To allow a comparison with hybrid and software prefetching, which only prefetch during the execution of Median's kernel, we also measured the impact when hardware prefetching is only enabled in the kernel. Table 6.7 shows the results of this experiment. For these simulations, we used a 128-entry shifted SPT and a write-validate cache. Without prefetching, the hit-ratio in Median's kernel is much lower than the overall hit-ratio due to the high rate of compulsory cache misses generated within the kernel. The hit-ratios improve when prefetching is activated but certainly not to the level of the hit-ratios in Table 6.5. To be more precise, we measured an MCPI improvement of 32% for SMedian when prefetching is allowed for the whole trace (see Table 6.5), while the MCPI only improves with 6.4% when strictly prefetching in the kernel (see Table 6.7). So, most of the performance gain due to SPT-based prefetching is obtained during the execution of auxiliary functions (e.g. the input and output routines) rather than in the kernel.

In Table 6.8, the results are shown for the early-prefetch optimisation. The table lists both the results for prefetching during the whole trace and kernel-only prefetching. Note, however, that the hit-ratios and MCPI values in the table are *overall* values (measured for the whole trace). As can be seen from Table 6.8, the early-prefetch optimisation is quite effective for the Median benchmark. It decreases the MCPI for kernel prefetching of SMedian with about 17% and for LMedian the MCPI is decreased with 25% compared to the normal prefetching mode. When comparing this to the Median execution without prefetching, this resolves into MCPI reductions of 23% and 31% respectively. If prefetching is allowed during the whole trace, then the improvements are even more substantial. In that case, the MCPI values have decreased with 48% (SMedian) and 60% (LMedian) compared to normal

Prefetch mode	SMedian				LMedian			
	Total trace		Kernel		Total trace		Kernel	
	Hit-ratio	MCPI	Hit-ratio	MCPI	Hit-ratio	MCPI	Hit-ratio	MCPI
Normal	98.53%	0.159	97.54%	0.219	99.08%	0.124	98.44%	0.146
Early	98.72%	0.085	97.74%	0.181	99.25%	0.049	98.62%	0.109

Table 6.8: Performance impact of the early-prefetch optimisation when using write-validate and a 128-entry shifted SPT.

SPT-based prefetching. This means that the MCPI values are reduced by 64% (SMedian) and 69% (LMedian) when comparing them with the no-prefetch results of Table 6.2. These large improvements indicate that, in the normal mode of prefetching, a lot of prefetches are started (or finished) too late.

So far, we did not mention the effect of prefetching on the bus utilisation. This is because the prefetching techniques we investigate are nearly perfect, implying that they issue a small number of erroneous prefetch requests. As a result, we measured only tiny differences ($< 1\%$) between the bus utilisation of non-prefetching and prefetching data caches.

Hybrid prefetching

For the hybrid prefetching experiments, we again use a cache model containing two prefetch blocks per set and which applies a write-validate policy. By default, we have parameterised the stream prefetch instructions with a lookahead of 3 and a stride of 8. This implies that prefetching runs $3 * 8 = 24$ bytes ahead. Remember that the following experiments should be compared with the kernel prefetching results of the previous section.

Table 6.9 shows the results of hybrid prefetching using PC-based synchronisation. The top entry of the table gives the performance of a non-prefetching data cache, while the remaining entries present the results for a prefetching cache with different PQ sizes. From Table 6.9 can be seen that, like for pure hardware prefetching, the PQ size only marginally affects the performance. A PQ of four entries seems to be the best choice when observing the MCPI values but the differences are only marginal.

When comparing these results to the ones in Table 6.8, one can conclude that hybrid prefetching yields lower MCPI values than normal, in-kernel hardware prefetching. More specifically, hybrid prefetching improves the MCPI on the average with about 13.5% compared to (normal) in-kernel hardware prefetching. However, hardware prefetching using the early-prefetch optimisation still outperforms hybrid prefetching for the Median traces when a runahead of 3 is applied.

In Table 6.10, the effects of varying the runahead are shown. Again, the first entry gives the results for a non-prefetching cache. The results indicate that the performance steadily improves when increasing the runahead until a runahead of 12 is reached. According to Table 6.10, a runahead of 12 yields the lowest MCPI values, which are even slightly lower than

Nr. of PQ entries	SMedian			LMedian		
	Overall hit-ratio	Hit-ratio in kernel	Overall MCPI	Overall hit-ratio	Hit-ratio in kernel	Overall MCPI
—	97.37%	95.80%	0.234	98.32%	95.80%	0.157
1	97.68%	98.89%	0.198	98.56%	98.88%	0.125
2	97.68%	98.89%	0.195	98.56%	98.88%	0.121
4	97.68%	98.89%	0.193	98.57%	98.89%	0.121
16	97.68%	98.89%	0.193	98.57%	98.89%	0.121

Table 6.9: Performance of hybrid prefetching (PC-synchronised, using a runahead of 3 references and write-validate).

Runahead	SMedian			LMedian		
	Overall hit-ratio	Hit-ratio in kernel	Overall MCPI	Overall hit-ratio	Hit-ratio in kernel	Overall MCPI
—	97.37%	95.80%	0.234	98.32%	95.80%	0.157
1	97.56%	97.64%	0.220	98.50%	97.98%	0.144
2	97.62%	98.26%	0.200	98.52%	98.31%	0.131
4	97.70%	99.09%	0.187	98.59%	99.18%	0.115
8	97.76%	99.69%	0.179	98.65%	99.91%	0.104
12	97.77%	99.80%	0.176	98.66%	99.99%	0.102
16	97.77%	99.78%	0.176	98.66%	99.97%	0.102

Table 6.10: Performance of hybrid prefetching (PC-synchronised, using a 16-entry PQ and write-validate).

the ones obtained by early-prefetch hardware prefetching (see Table 6.8). It is not surprising that both hybrid prefetching with a runahead of 12 and early-prefetch hardware prefetching perform almost identically: both methods more or less guarantee that the succeeding cache block is prefetched as soon as its predecessor is referenced. For example, a runahead of 12 means a prefetch distance of $8 * 12 = 96$ bytes, which is larger than the 64-byte block-size of the cache. So, from Table 6.10 can be concluded that a reasonably large runahead is essential for effective prefetching in the Median benchmark.

Table 6.11 presents the results of data-synchronised hybrid prefetching with different PQ sizes. It may be surprising that data-synchronised prefetching performs worse than PC-synchronised prefetching for a 1-way PQ because the latter type of prefetching may flood the PQ more easily. The poorer performance of data-synchronised prefetching is due to the fact that if the cache cancels a prefetch request, then the particular cache block can not be prefetched anymore as there is only one prefetch request per cache block in data-synchronised prefetching (see Section 6.3.1). This is in contrast to the PC-synchronised mode, in which multiple prefetch requests may be issued for one cache block. Therefore,

Nr. of PQ entries	SMedian			LMedian		
	Overall hit-ratio	Hit-ratio in kernel	Overall MCPI	Overall hit-ratio	Hit-ratio in kernel	Overall MCPI
—	97.37%	95.80%	0.234	98.32%	95.80%	0.157
1	97.68%	98.00%	0.205	98.47%	97.70%	0.132
2	97.68%	98.91%	0.192	98.57%	98.90%	0.120
4	97.68%	98.91%	0.192	98.57%	98.90%	0.120
16	97.68%	98.91%	0.192	98.57%	98.90%	0.120

Table 6.11: Performance of hybrid prefetching (data-synchronised, using a runahead of 3 and write-validate).

Runahead	SMedian			LMedian		
	Overall hit-ratio	Hit-ratio in kernel	Overall MCPI	Overall hit-ratio	Hit-ratio in kernel	Overall MCPI
—	97.37%	95.80%	0.234	98.32%	95.80%	0.157
1	97.56%	97.63%	0.218	98.50%	97.99%	0.142
2	97.66%	98.68%	0.201	98.55%	98.65%	0.131
4	97.71%	99.15%	0.187	98.60%	99.27%	0.114
8	97.76%	99.71%	0.179	98.65%	99.92%	0.104
12	97.76%	99.73%	0.177	98.66%	99.99%	0.102
16	97.76%	99.73%	0.177	98.66%	99.99%	0.102

Table 6.12: Performance of hybrid prefetching (data-synchronised, using a 16-entry PQ and write-validate).

cancellations are less critical in this mode. Apparently, flooding of the PQ is not always a problem but it may have positive effects as well.

The results from Table 6.11 suggest that when increasing the PQ size (even to a PQ size of 2) the critical cancellations are avoided. In these cases, the obtained MCPI values are roughly identical to the ones of PC-synchronised prefetching. The small variations between the results of PC and data synchronised prefetching are caused by the different strategies for issuing prefetch requests to prevent PQ flooding.

Table 6.12 shows the impact of runahead on data-synchronised prefetching. As these results are only marginally different from the PC-synchronisation results as listed in Table 6.10, one can conclude that both PC-synchronised and data-synchronised prefetching are affected in the same manner when changing the runahead.

6.4.2 Hybrid data-synchronised versus scalar prefetching

In this section, we study the performance differences between pure software (i.e. scalar) prefetching and hybrid data-synchronised prefetching. The scalar prefetching was implemented using the straightforward technique of having a prologue which establishes the runahead, the (kernel) loop itself and an epilogue, being the last iterations of the loop without prefetching (see Figure 6.3). No effort was made to calibrate the scalar prefetch instructions such that they only refer to the start of a cache block in order to limit the number of prefetch instructions to the ones that are essential. Because the multiple streams are misaligned with respect to each other, this would require a major, and probably an awkward, rewrite of the application.

Again, we used the Median benchmark as workload. However, for the following experiments, we only simulated the SMedian trace (de-interlacing a 400$\times$20 image). Moreover, the results for hybrid prefetching in the following study slightly differ from the hybrid prefetching results presented in the previous section. This unfortunate discrepancy is due to several changes that have been made to the TriMedia compiler during the period between the two studies. For example, the number of instruction fetch and data reference operations in the SMedian traces used in the two studies differ significantly. Therefore, a comparison

Cycles_{load}	MCPI_{load}	Hit-ratio	# Bank cfl.
7.1%	0.419	97.28%	3973

Table 6.13: Performance results for SMedian without prefetching.

between the results of both studies must be performed with care.

In the following study, we have only used the write-validate cache policy. This implies that prefetching is only done with respect to load operations. For this reason, we do not use the generic MCPI metric but, instead, we use a MCPI_{load} metric. This MCPI_{load} is the CPI fraction due to cache penalties from load operations only. Formally,

$$\text{MCPI} = \frac{\text{MCPI}_{load} + \text{MCPI}_{store}}{2}$$

Additionally, we introduce a new metric, called *Cycles*, which denotes the percentual performance improvement of the total cycle count due to prefetching when compared to simulation without prefetching. For this metric, a negative value means a speedup while a positive value means a slowdown.

Table 6.13 shows the performance results of SMedian when no prefetching is performed. The $Cycles_{load}$ metric denotes the fraction of the total number of cycles that is wasted due to cache penalties from load operations. So, the Cycles_{load} suggests that only 7.1% of the total cycle count can be improved with latency reduction techniques, such as prefetching (note that we only prefetch on loads as we apply the write-validate policy). In other words, a reduction of 7.1% of the total cycle count would mean that there are no load penalties anymore (this includes the entire trace, so also the auxiliary routines). Moreover, Table 6.13 also presents the number of bank conflicts that occurred in the data cache during the simulation. The relevance of this metric will be shown later.

In Table 6.14, the performance results are presented when applying either scalar or hybrid prefetching. Interestingly enough, scalar prefetching only obtains speeddowns. Apparently, the overhead induced by the prefetch instructions is too large to benefit from the prefetching itself. This effect is, of course, intensified due to the fact that our benchmark issues multiple (scalar) prefetch requests for the same cache block. Another interesting point is the number of bank conflicts. The compiler seems to have more freedom of scheduling memory references when prefetch instructions are included in the loop, thereby reducing the number of bank conflicts.

For hybrid prefetching, all experiments obtain speedups. Note that a speedup of 1.8% means that $\frac{1.8}{7.1} = 25\%$ of Cycles_{load} is reduced. Moreover, the runahead of the stream prefetch instructions is not really affecting the results. This is in contrast to the hybrid prefetch results presented in the previous section where the runahead *did* impact the performance quite substantially. This difference is caused by the new compiler used for the experiments in this study. The SMedian trace generated for this new compiler contains more instruction fetches and less data references in the kernel. Hence, the kernel consists of more instructions that do not access the data cache, implying that the amount of computational work per kernel loop iteration has increased. As a consequence, the implicit prefetch distance (i.e. the runahead) of the kernel loop has become larger, thereby diminishing the effect of the explicit runahead

	Runahead	Cycles	MCPI$_{load}$	Hit-ratio	# Bank cfl.
Scalar	1	+2%	0.250	97.70%	211
	2	+1.6%	0.230	97.73%	249
	4	+1.4%	0.224	97.75%	325
	8	+1.3%	0.232	97.70%	477
	16	+1.1%	0.239	97.73%	781
Hybrid	1	-1.7%	0.306	97.69%	3982
	2	-1.8%	0.298	97.71%	3982
	4	-1.8%	0.298	97.71%	3982
	8	-1.8%	0.297	97.71%	3982
	16	-1.8%	0.297	97.71%	3982

Table 6.14: Performance results of scalar and hybrid prefetching for the SMedian benchmark.

established by the stream prefetch instruction.

The obtained values for MCPI$_{load}$ might, at first sight, be somewhat surprising as they are lower for scalar prefetching than for hybrid prefetching. The lower MCPI$_{load}$ values for scalar prefetching are due to the large number of extra prefetch instructions. The additional execution time of these instructions reduces the relative fraction of time during which the processor stalls due to cache penalties. In other words, the extra prefetch instructions increase the I_{total} in the MCPI formula (see Section 6.4.1), thereby reducing the MCPI.

In the following experiment, of which the results are shown in Table 6.15, we unrolled Median's innerloop 2 or 4 times. In the case of scalar prefetching, we kept the number of prefetch instructions per loop iteration constant while unrolling the loop. This means that the total number of scalar prefetch instructions has decreased for the loop-unrolled versions of Median (as there are less iterations and still the same number of prefetch instructions per

	Unroll factor	Cycles	MCPI$_{load}$	Hit-ratio	# Bank cfl.
No pref.	0	—	0.419	97.28%	3973
	2	—	0.421	97.22%	3972
	4	—	0.431	97.27%	3973
Scalar	0	+1.1%	0.239	97.73%	781
	2	-0.5%	0.239	97.72%	781
	4	-2.5%	0.231	97.73%	781
Hybrid	0	-1.8%	0.297	97.71%	3982
	2	-2.1%	0.294	97.75%	3982
	4	-2.6%	0.288	97.73%	3982

Table 6.15: Performance results of scalar and hybrid prefetching when loop-unrolling is applied. All results are for a runahead of 16.

	# PQ entries	Cycles	MCPI$_{load}$	Hit-ratio
Scalar	1	-2.3%	0.238	97.71%
	2	-2.4%	0.231	97.73%
	4	-2.5%	0.231	97.73%
	8	-2.5%	0.231	97.73%
	16	-2.5%	0.231	97.73%
Hybrid	1	-1.2%	0.359	97.59%
	2	-2.5%	0.293	97.72%
	4	-2.5%	0.291	97.72%
	8	-2.6%	0.288	97.73%
	16	-2.6%	0.288	97.73%

Table 6.16: Performance results when the prefetch queue size is varied. The runahead in this experiment is 16 and the kernel loop is unrolled 4 times.

loop iteration). As a consequence, the overhead of the prefetch instructions should have been reduced.

Table 6.15 clearly shows that loop unrolling is essential for scalar prefetching, that is, for our scalar prefetching implementation. Unrolling the loop 4 times results in a performance gain which is identical to the one obtained by hybrid prefetching: about 2.5%. This is a $\frac{2.5}{7.1}$ = 35% reduction of the Cycles$_{load}$ from Table 6.13. This improvement is partly due to the reduction of the number of scalar prefetch instructions. But as the speedup for hybrid prefetching also improves when unrolling the kernel loop, there is an additional effect causing this performance improvement. The loop unrolling has made the code more efficient while the cache penalties have remained unchanged. Thus, in other words, the performance of Median has become increasingly memory bound. This means that a reduction of the number of cache misses due to prefetching will have a greater impact on the total number of executed cycles when compared to the non-unrolled Median version.

In Table 6.16, the results are shown when the size of the PQ is varied. In this experiment, the innerloop is unrolled 4 times. Scalar prefetching is not really affected by the PQ size. This is a consequence of the fact that in this type of prefetching a lot of prefetch requests for the same cache block are issued. When one of them is cancelled due to a full PQ, there is a high probability another prefetch request will take over. In accordance to what was found in the previous section (the study of hardware versus hybrid prefetching), hybrid prefetching with a prefetch queue of only 1 entry yields poor performance. Again, this is due to the issuing of one prefetch request per cache block. These requests may easily be cancelled when using a single-entry PQ, implying that the cache block in question cannot be prefetched anymore. As can be seen from Table 6.16, increasing the PQ size to 2 or more entries alleviates this problem.

The final experiment focuses on hybrid prefetching only. In Table 6.17, the cache performance is shown when the processor exploits subword or SIMD parallelism. With special SIMD instructions, operations on several (distinct) data elements, which are packed into a single word, can be performed simultaneously. Assume, for example, that a data word

	# PQ entries	Cycles	MCPI_{load}	Hit-ratio
No pref.	—	0%	0.431	97.27%
Hybrid	1	-1.7%	0.364	97.59%
	2	-3.5%	0.292	97.72%
	4	-3.5%	0.290	97.72%
	8	-3.5%	0.289	97.73%
	16	-3.5%	0.289	97.73%

Table 6.17: Performance results when applying SIMD operations. In this experiment, the runahead is 16 and the innerloop is unrolled 4 times.

contains four pixel values and that the special SIMD instructions operate on 4-tuples such that the calculations are done on each element of the 4-tuple individually. For instance, the *quadadd* instruction would perform $(a, b, c, d) + (p, q, r, s) \rightarrow (a + p, b + q, c + r, d + s)$.

As this type of parallelism is well suited for imaging applications, future TriMedia versions will feature a set of SIMD instructions. To investigate the impact of SIMD parallelism on data prefetching, we filtered the address trace such that a number of instruction-fetch events belonging to arithmetic operations were removed. This removal mimics the combination of several arithmetic operations into one SIMD instruction. Evidently, this method is not accurate and the results should therefore be interpreted with care. However, the experiment gives us a rough feeling of how the performance is affected when more powerful arithmetic operations are applied.

The results in Table 6.17 indicate that prefetching becomes more important when SIMD parallelism is exploited (the reduction of the total number of cycles has increased to -3.5%). The reason for this is identical to the effect of unrolling the kernel's loop. More specifically, the SIMD operations improve the computational efficiency of the program, while the memory latencies remain unchanged. As a consequence, the program's performance has become increasingly memory bound.

6.5 Validating our model

Struik et al. [134] presented hybrid prefetching experiments which are similar to the ones discussed in this study. However, Struik's study uses the cycle-accurate TMsim simulator [124] rather than our more abstract, but faster and more flexible, operation-driven simulator. Nevertheless, the similarity between their experiments and the ones presented in this chapter provides us with a means to validate our 1-CPI model results against the accurate TMsim results. So, where it is possible, we will indicate the differences (or the similarities, for that matter) between the two studies.

In Table 6.18, the performance estimations of both the cycle-accurate TMsim simulator and our operation-driven simulator are shown when varying the PQ size. Columns two and three give the estimated cycle count for the processing of one video frame for TMsim and Mermaid respectively. The fourth column shows the error between these two estimations. In columns five and six, the number of executed prefetches for both simulator is given. Fi-

Nr. of	Cycles_{1frame}		Error	# prefetches		Hit-ratio_{loads}	
PQ entries	TMsim	Mermaid	Cycles_{1frame}	TMsim	Mermaid	TMsim	Mermaid
8	23683	22324	5.7%	234	244	100%	99.98%
4	23683	22396	5.4%	234	240	100%	99.91%
2	23684	22426	5.3%	235	238	100%	99.90%
1	24379	25767	5.7%	118	121	97.95%	97.81%

Table 6.18: Validating our results against the TMsim cycle-accurate simulation results [134] when varying the PQ size. In this experiment, the loop is 4 times unrolled and the runahead is 16.

Unroll	Cycles_{1frame}		Error	# prefetches		Hit-ratio_{loads}	
factor	TMsim	Mermaid	Cycles_{1frame}	TMsim	Mermaid	TMsim	Mermaid
0	33313	31868	4.3%	234	244	100%	99.98%
2	25862	24531	5.1%	234	243	100%	99.96%
4	23683	22324	5.7%	234	243	100%	99.96%

Table 6.19: Validating our results for loop unrolling against the TMsim cycle-accurate simulator. In this experiment, the prefetch queue has 8 entries and the runahead is 16.

nally, the last two columns present the hit-ratios for load operations while prefetching is activated (kernel hit-ratio).

The relatively small differences between the results for both simulators suggests that our simulation model is fairly accurate. More important, although the absolute numbers obtained by our model slightly differ from the ones obtained by TMsim, the trends are identical for both simulators, indicated by a fairly constant error (5$\pm$0.7 %).

Table 6.19 shows the hybrid prefetching outcomes of both simulators when applying loop unrolling. The meaning of the columns is identical to that of the previous table. Again, the errors are relatively small and the behaviour of Cycles_{1frame} is identical for both simulators. So, as was already shown in Muller's dissertation [97], a reasonably abstract simulation model may already give good performance estimates. These validation results are of great importance as they increase our confidence in the obtained simulation results.

Finally, in the TMsim study from [134], Struik et al. also performed an experiment which is similar to our SIMD experiment. However, that experiment applied real SIMD instructions rather than mimicking their presence by removing instruction-fetch events. The cycle-accurate results show that the trend reflected by the results of our simple SIMD experiment (as were shown in Table 6.17) is quite realistic as TMsim also indicates that prefetching becomes more important when applying SIMD parallelism.

6.6 Discussion

In this case study, we discussed the concept of data prefetching with the intention to improve the average memory latency of future Philips TriMedia processors. We first presented a classification of the different types of data prefetching, ranging from pure software prefetching to pure hardware prefetching. From this prefetching classification, we concluded that a hybrid software/hardware prefetching technique yields the highest degree of flexibility and transparency while the prefetch engine is still straightforward and cheap to implement.

Many hardware and hybrid prefetching techniques synchronise the issuing of prefetch requests on the instruction address of data references. We have shown that this synchronisation method has several drawbacks. For this reason, we have proposed a novel hybrid prefetching technique which synchronises on data addresses rather than on the program counter. This may increase the robustness of the prefetch engine considerably.

To investigate the performance impact of data prefetching for the TriMedia architecture, we have performed a simulation study using Mermaid. We studied most of the prefetching techniques identified in our classification, including our newly proposed prefetching method. Unfortunately, we were able to study one benchmark program only. Evidently, this reduces the value of the obtained results with respect to their absolute meaning (we cannot extrapolate the results to the general case). For this reason, we used the benchmark mainly to investigate the relative differences between the various prefetching techniques.

A number of observations can be made from the results of our experiments. First, the results indicate that prefetching is an effective latency reduction technique for the workload we have studied. We measured reductions of up to roughly 70% of the CPI fraction due to cache penalties.

A second observation is that pure hardware prefetching obtains the largest improvements for the applied workload. This is because hardware prefetching allows for prefetching during the execution of the whole application. By contrast, hybrid and software prefetching only allow for prefetching during code segments which were explicitly instrumented to do so. As a consequence, the code segments which would not have been instrumented for prefetching in the case of software or hybrid prefetching (e.g. system calls) may still benefit from hardware prefetching.

When focusing on the code segments in which prefetching is enabled for all evaluated techniques, we found the hybrid prefetching technique to be the most effective one. Nevertheless, the performance differences of the various prefetching techniques are minimal. But, as hybrid prefetching is more transparent (and thus easier in its use) than pure software prefetching and cheaper to implement compared to pure hardware prefetching, we clearly prefer hybrid prefetching.

With respect to the different types of hybrid prefetching, we have shown that our proposed method obtains a performance gain which is identical to that of the more traditional hybrid prefetching techniques (using synchronisation on the program counter). However, we believe that our method is easier to implement and more robust than the traditional techniques.

For several experiments, we were able to validate our results against those from a cycle-accurate TriMedia simulator. The validation results indicate that the differences in predicted cycle-count are reasonably small (roughly 5%). Furthermore, all trends we measured are identical for both simulators. Evidently, this increases our confidence in the obtained results.

Equally important is the fact that we obtain these results in a fraction of the time needed by the cycle-accurate simulator (our current Mermaid simulator, which is not yet optimised for speed, is about an order of magnitude faster than the cycle-accurate TriMedia simulator), allowing us to explore a wider design space than which is possible with the cycle-accurate simulator. To conclude, the ease with which we constructed our simulation model and its good simulation performance, together with the encouraging validation results, demonstrate that Mermaid's simulation framework is suitable for zooming in on the evaluation of uni-processor platforms.

Chapter 7

Evaluation of LH*LH for a multicomputer architecture

"I have traveled the length and breadth of this country and talked with the best people, and I can assure you that data processing is a fad that won't last out the year"

The editor in charge of business books for Prentice Hall, 1957

Whereas the two previous case studies mainly focused on the performance evaluation at the architecture level, this chapter presents a case study in which the emphasis is on the performance evaluation at the application level. This should demonstrate that Mermaid is capable of evaluation at both the architecture and application levels. The application we study in this chapter is strongly related to distributed database systems. Clearly, this type of application is different from the workloads we have used so far, which were mostly related to scientific computing.

Modern database applications require fast access to large volumes of data. Sometimes the amount of data is so large than it cannot be efficiently stored or processed by a uniprocessor system. Therefore, a *distributed data structure* can be used that distributes the data over a number of processors within a parallel system (or a number of workstations in a local area network, for that matter). This is an attractive possibility because the achievements in the field of communication networks for parallel and distributed systems have made remote memory accesses faster than accesses to the local disk [52]. So, even when disregarding the additional processing power of parallel platforms, it has become more efficient to use the main memory of other processors than to use the local disk.

There are many ways in which the data can be distributed over multiple processors. The simplest variant is *striping*, which implies that the data is partitioned across the processors in a round-robin fashion. Alternatively, data records can be distributed according to their keys using a hash function or a range partitioning. The drawback of all these techniques is that they are static and thus not scalable. They do not allow the data to easily expand over more processors than there were initially allocated. Expansion of the data requires an explicit redistribution of all data and cannot be done in a scalable, incremental way. However, scalable data structures are highly desirable in modern database systems. The database's storage requirements may easily grow beyond the processor's physical memory limit, which may result in a severe performance degradation due to the extra swapping, page misses etc. A scalable data structure can be characterised as follows [66]:

- The time to insert and retrieve data is independent of the number of stored data elements.
- The data structure can handle any amount of data; there is no theoretical upper limit after which performance degrades.
- The data structure grows and shrinks incrementally rather than it has to reorganise itself totally on a regular basis (e.g. rehashing of all distributed records).

For distributed memory parallel systems, a number of Scalable Distributed Data Structures (SDDSs) have been proposed which provide the above features [85]. In these distributed storage methods, the processors are divided into *clients* and *servers*. A client manipulates the data by inserting data elements, searching for them or removing them. A server stores a part of the data, called a *bucket*, and receives requests from clients that operate on the bucket. Generally, there are three ground rules for the implementation of an SDDS in order to realise a high degree of scalability [66]:

- The SDDS should not be addressed using a central directory which forms a bottleneck.
- Each client should have an image of how data is distributed which is as accurate as possible. This image should be improved each time a client makes an "addressing error", i.e. contacts a server which does not contain the required data. The client's state (its current image) is only needed for efficiently locating the remote data; it is not required for the correctness of the SDDS's functionality.
- If a client has made an addressing error (due to an outdated image), then the SDDS is responsible for forwarding the client's request to the correct server and for updating the client's image.

For an efficient SDDS, it is essential that the communication needed for data operations (retrieval, insertion, etc.) is minimised while the the amount of data residing at the server nodes (i.e. the *load factor*) is well balanced. In [85], Litwin et al. propose an SDDS, called *LH**, which addresses the issue of low communication overhead and balanced server utilisation. This SDDS is a generalisation of Linear Hashing (LH) [83, 84], which will be elaborated upon in the next section. For LH*, insertions usually require one message (from client to server) and three messages in the worst case. Data retrieval requires one extra message as the requested data has to be returned.

In this case study, we evaluate the performance of a variant of the LH* SDDS, called LH*LH. For this purpose, we use a simulation model which is based on the architecture of a Parsytec CC multicomputer but which can be configured to simulate several different types of network topologies. With this model, we investigate how scalable the LH*LH SDDS actually is and which factors affect the scalability of this particular SDDS. Our interest in the LH*LH SDDS originates from the fact that this data structure will eventually form the heart of a parallel version of the Monet database system [15]. As part of the IMPACT project [57], an initial version of this parallel database is developed for a Parsytec CC multicomputer.

7.1 Linear hashing

Linear Hashing (LH) is a method to dynamically manage a table of data. More specifically, it allows the table to grow or shrink in time without suffering from a penalty with respect to the space utilisation or the access time. The LH table is formed by $N \times 2^i + n$ buckets, where N is the number of starting buckets ($N \geq 1$ and $n < 2^i$). The meaning of i and n is explained later on. The buckets in the table are addressed by means of a pair of hashing functions h_i and h_{i+1}, with $i = 0, 1, 2...$ Each bucket can contain a predefined number of data elements. The function h_i hashes data keys to one of the first $N \times 2^i$ buckets in the table. As we show later on, the function h_{i+1} is used to hash data keys to the remaining buckets. A popular way to hash the data keys is by using a modulo-based hash function:

$$h_i(\text{key}) \rightarrow \textit{key mod } (N \times 2^i) \tag{7.1}$$

The LH data structure grows by splitting a bucket into two buckets whenever there is a collision in one of the buckets. With a collision, we mean that a certain load threshold is exceeded. To illustrate this bucket splitting, consider Figure 7.1. In this figure, we assume that N equals to 2, that the hash functions are of the form as shown in Equation 7.1 and that buckets have a load threshold of two elements. At the moment a collision occurs (the insertion of 4 in Figure 7.1a), a bucket has to split. Which bucket has to be split is determined by a special pointer, referred to as n (pointing to bucket 0 in Figure 7.1a). So, even in the case we insert a number into bucket 1 (which is also causing a collision), bucket 0 is still split. The actual splitting involves three steps: creating a new bucket, dividing the data elements over the old and the newly created bucket and updating the pointer n. Dividing the data elements over the two buckets is done by applying the function h_{i+1} to each element in the splitting bucket. The n pointer is updated by applying $n = (n + 1)\ mod\ N \times 2^i$. Figure 7.1b shows the situation after bucket 0 has split. Because of the splitting, indexing the LH data structure is performed using both h_i and h_{i+1}. Or, formally:

$$\begin{aligned} \text{index}_{bucket} &= h_i(\text{key}) \\ \text{if } (\text{index}_{bucket} < n) \text{ then } \text{index}_{bucket} &= h_{i+1}(\text{key}) \end{aligned} \tag{7.2}$$

As the buckets below the n pointer have been split, these buckets should be indexed using h_{i+1} rather than with h_i. Figure 7.1c shows that inserting a key with number 5 again causes a split to occur. This time, bucket 1 needs to be split. After this split, n wrapped around to 0. When this happens, i (which is often called the *bucket-level*) should be incremented.

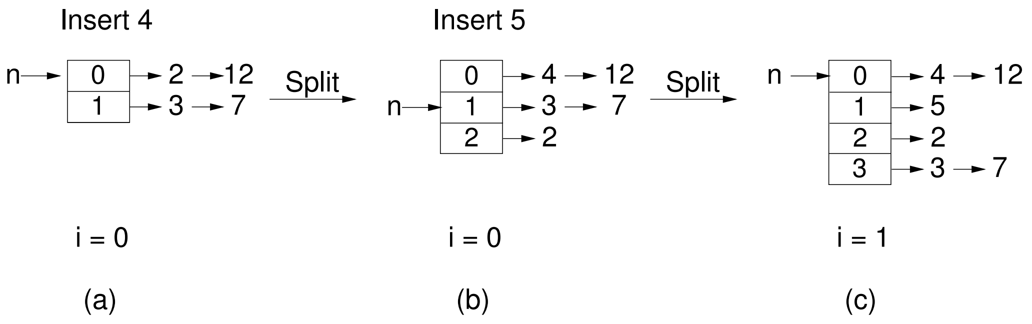


Figure 7.1: The splitting concept in linear hashing.

So far, we assumed that a bucket splits whenever a collision occurs (i.e. a fixed load threshold is exceeded). This is called *uncontrolled splitting*. However, one could also initiate a split whenever the load factor of an LH bucket has exceeded a dynamic threshold value which is based on the global load factor including all buckets. It has been shown that this method, which is called *controlled splitting*, results in a better utilisation of the buckets within the LH data structure [85].

The process of shrinking is similar to the growing of the LH data structure. Instead of splitting buckets, two buckets are merged whenever the load factor drops below a certain threshold. In this case study, we limit our discussion to the splitting within SDDSs. Furthermore, we assume that the hash functions are always of the form as shown in Equation 7.1.

7.2 The LH*LH SDDS

The LH* SDDS is a generalisation of linear hashing (LH) to a distributed memory parallel system [85]. In this case study, we focus on one particular implementation variant of LH*, called LH*LH [66, 67]. The LH*LH data is stored over a number of server processes and can be accessed through dedicated client processes. These clients, which are not part of the actual LH*LH SDDS, form the interface between the application and LH*LH. We assume that each server stores *one* LH*LH bucket of data, which implies that a split always requires the addition of an extra server processes. This scheme could, of course, be optimised by placing multiple LH*LH buckets at a single server. Globally, the servers apply the LH* scheme to manage their data, while the servers use traditional LH for their local bucket management. Thus, a server's LH*LH bucket is implemented as a collection of LH buckets. Hence, the name LH*LH. In Figure 7.2, the concept of LH*LH is illustrated.

As was explained in the previous section, addressing a bucket in LH is done using a

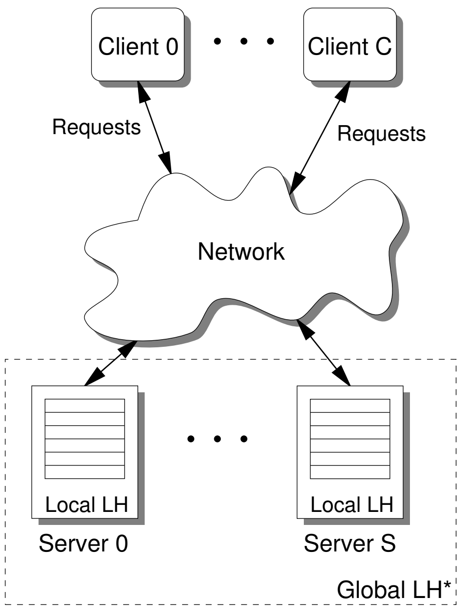


Figure 7.2: The LH*LH SDDS.

key and the two variables i and n (see Equation 7.2). In LH*LH, the clients address the servers in the same manner. To do so, each client has its own *image* of the values i and n: i' and n' respectively. Because the images i' and n' may not be up to date, clients can address the wrong server. Therefore, the servers need to verify whether or not incoming client requests are correctly addressed, i.e. can be handled by the receiving server. If an incoming client request is incorrectly addressed, then the server forwards the request to the server that is believed to be correct. For this purpose, the server uses a forwarding algorithm [85] for which it is proven that a request is forwarded at most twice before the correct server is found. Each time a request is forwarded, the forwarding server sends a so-called Image Adjustment Message (IAM) to the requesting client. This IAM contains the server's local notion of i and n and is used to adjust the client's i' and n' in order to get them closer to the global i and n values. As a consequence, future requests will have a higher probability of being addressed correctly.

The splitting of an LH*LH bucket is similar to the splitting of LH buckets. The pointer n is implemented by a special token which is passed from server to server in the same manner as n is updated in LH: it is forwarded in a ring of the servers 0 to $N \times 2^i$, where N is the number of starting servers. When a server holds the n token and its load factor is larger than a particular threshold, the server splits its LH*LH bucket and forwards the n token. Splitting the LH*LH bucket is done by initialising a new server (by sending it a special message) and shipping half of its LH buckets to the new server (remember that the LH*LH bucket is implemented as a collection of LH buckets). The use of LH local at the servers in combination with global LH* allows for efficient splitting of the data. By sharing the hashed keys between the local LH and the global LH* schemes, it is not required to visit or rehash all data elements when splitting the server's data. Instead, the LH buckets that have an odd index are shipped to the new server while the even buckets are compacted and stay at the splitting server. This is illustrated in Figure 7.3. A comprehensive explanation of how the sharing of keys between LH and LH* works, can be found in [67].

It has been shown in [85] that a splitting threshold which can be dynamically adjusted

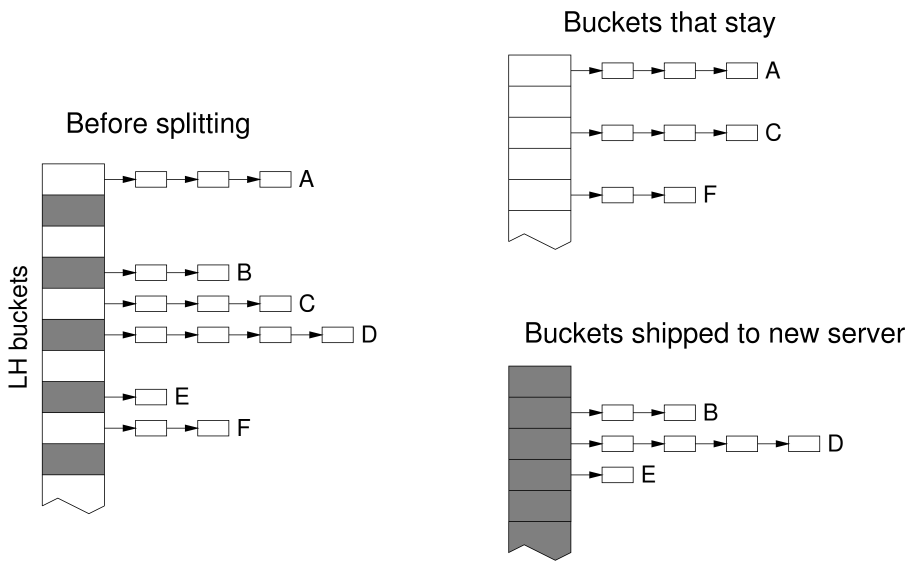


Figure 7.3: Splitting the local LH buckets of an LH*LH bucket.

performs better than a static one. Therefore, LH*LH applies a dynamic threshold which is based on an estimation of the global load factor and which is calculated by the following formula [67]:

$$T = M \times V \times \frac{2^i + n}{2^i}$$

where M is a sensitivity parameter and V is the capacity of an LH*LH bucket in number of data elements. Typically, M is set to a value between 0.7 and 0.9. If the number of data elements residing at the server with the n token is larger than the threshold T, then the server splits its data. We note that in this scheme the decision to split is made autonomously, which means that an underloaded token-holder can keep other overloaded servers from splitting.

In LH*LH, splits are performed in a concurrent manner which allows the server to continue with handling client requests. If the server notices that a requested piece of data has been or is being shipped, then it forwards the request to the new server. In the case the requested data is being shipped, the server takes care of the correct serialisation: first the shipment has to be finished before the request can be forwarded. A more detailed description of the LH*LH data structure can be found in [66, 67].

In the next section, we describe how we used Mermaid as a vehicle to study the scalability behaviour of LH*LH for a multicomputer architecture. Thereafter, we present several experiments in which we investigate the factors that affect LH*LH's scalability and study its interaction with the multicomputer's architecture.

7.3 The simulation model

The parallel architecture we focus on in this study is based on that of a Parsytec CC multicomputer. The platform's configuration we have used for our experiments consists of 16 PowerPC 604 processors connected in a network with an application-level throughput of 27 MByte/s for point-to-point communication. The network topology of this particular machine, which has been derived from the Clos topology (see Chapter 5), is shown in Figure 7.4. In this evaluation study, we focus on the communication load that is generated by the LH*LH SDDS. So, to model this multicomputer architecture, we have taken the communication model from Chapter 5 (see Figure 5.6) and configured it to support the topology of the Parsytec CC. For performance reasons, we decided to embed the server and client

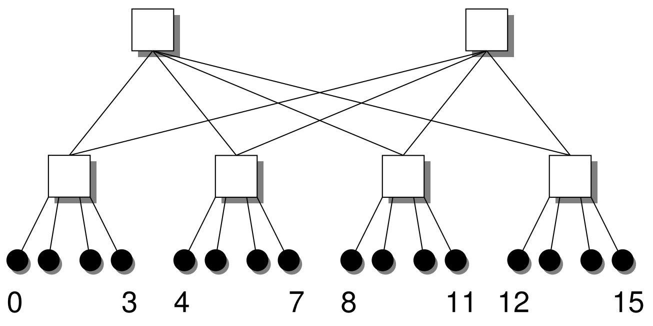


Figure 7.4: The network topology of the modelled Parsytec CC. The white boxes refer to routers and the black circles to processors.

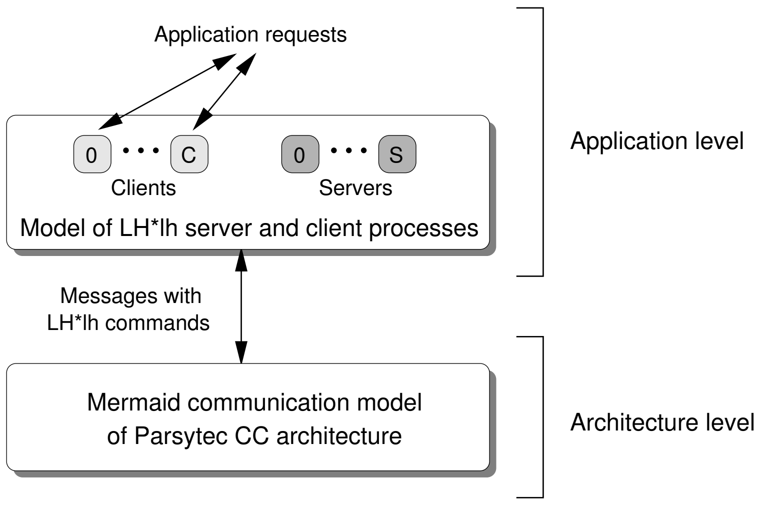

Application requests
0 ··· C
Clients
0 ··· S
Servers
Model of LH*lh server and client processes
Application level
Messages with LH*lh commands
Mermaid communication model of Parsytec CC architecture
Architecture level

Figure 7.5: The simulation model for studying the LH*LH SDDS.

processes, which are part of the application model, into the architectural communication model. In the resulting monolithic model, we do not need an explicit interface between the application and architecture levels (like the interfaces that have been discussed in Section 3.1.5) which slows down the simulation. To embed the clients and servers into the architectural communication model from Chapter 5, we substituted the processor component of this model with a component that models either an LH*LH server or client process. This implies that a modelled node can contain one server or client process only. Another consequence is that the clients are always placed on the multicomputer's nodes. So, we are not able to evaluate alternative schemes in which the clients are placed on, for example, a fast front-end machine of the multicomputer.

Although there are some limitations regarding the placement of server and client processes, the ease with which we constructed the simulation model again illustrates the flexibility that is achieved with Mermaid. By simply exchanging a single model component (i.e. the processor), which can be done with a relatively small amount of effort, we are able to study LH*LH's performance behaviour.

In Figure 7.5, the infrastructure of our simulation model is shown. For the sake of convenience, the figure separates the application and architecture levels. But, like we said before, in reality these levels are integrated into one (communication) model. As illustrated in Figure 7.5, a particular application can access the distributed data via the client processes. In this study, we investigate an application which builds the distributed data structure by using a Dutch dictionary as the data keys. The dictionary contains roughly 180,000 words.

The client processes provide the architecture level of the simulation model with communication requests. These requests are messages from the client to a server containing commands that operate on the LH*LH SDDS (e.g. insert, lookup, etc.). The server processes also issue communication requests, like when a client request has to be forwarded to another server. In Table 7.1, a complete overview of all possible communication requests is given.

The insert, lookup and remove requests are more or less self-explanatory. For the insertion request, the *Datasize* parameter refers to the size of the data element that is inserted.

Type of request	Parameters	From	To
Insert	Key, Datasize	Client	Server
Insert (forward)	Key, Datasize	Server	Server
Lookup	Key	Client	Server
Lookup (forward)	Key	Server	Server
Remove	Key	Client	Server
Remove (forward)	Key	Server	Server
Token forward	—	Server	Server
Shipment	Keys, Datasize	Server	Server
Start new server	i	Server	Server
Image Adjustment Message (IAM)	i, n	Server	Client
Acknowledgement	Null or Datasize	Server	Client

Table 7.1: Types of communication requests in the LH*LH model.

The token is forwarded with a special token forward request. This request does not contain a parameter. The shipment request indicates the shipment of *one LH bucket* from the splitting server to a newly created server. Its parameters specify the keys of the data elements that are being shipped and the total size of the shipped data. In our model, the data of one LH bucket is shipped using a single bulk message. Before a server can split its data, a new server has to be created by sending it a special "activation" request. This request contains the bucket-level i as parameter. To adjust the clients' image of i' and n', there is an IAM request which is sent from server to client. Finally, all client requests are acknowledged in our model of LH*LH. For insertions and removals, the acknowledgement is just an empty message and for lookups the acknowledgement contains the size of the returned data element.

The architecture level of the communication model models a wormhole-routed network that simulates the communication requests at the flit-level, like was explained in Chapter 5. The model can be configured to simulate the Parsytec CC's topology (as shown in Figure 7.4), the generic Mesh of Clos topology or a mesh topology. For routing, the model uses the same scheme as in Section 5.2.1 (deterministic XY-routing for mesh networks and a deterministic scheme based on the source node's identity for multistage networks). Furthermore, messages larger than 4K are modelled to be split up in separate packets of 4K each.

7.3.1 Validation

To validate the architecture level of our simulation model, we have performed several validation experiments using the message-roundtrip benchmark and the stress-testing benchmarks from Section 4.1. The benchmarks were both executed on the real Parsytec CC and simulated by our model after which the real and predicted execution times were compared. For all experiments, we measured an average error that does not exceed 3.5%. The worst-case error that was measured equals to 9.8%. Unfortunately, several of the simulations obtain an average error with a reasonably large standard deviation. This is especially true for the message-roundtrip benchmark results. For this particular experiment, we measured an average error of 3.4% and a standard deviation of 4.1. The high variance is probably caused by the AIX kernel that runs on the nodes of the real machine. Measurements on the real machine, for example, show an irregular performance behaviour for simple message transfers. So far, we have not been able to capture this irregular behaviour in our abstract communication model.

7.4 Experiments

We performed a range of experiments with an application which builds the LH*LH SDDS using a Dutch dictionary. By default, our architecture model only accounts for the delays that are associated with communication. Computation performed by the clients and servers is not modelled. In other words, the clients and servers are infinitely fast. This should give us an upper bound of the performance of LH*LH when using the modelled network technology. We believe that the assumption of constant server latencies (which are in our case, by default, zero) still gives realistic results. This is because linear hashing is used for local storage at the servers, for which insertions are $O(1)$ on the average. Furthermore, the data elements that are being inserted consist of a key only (they do not have “a data body”) unless stated otherwise. Throughout this section, we use the term *blobsize* when referring to the size of the body of data elements (so, by default, the blobsize is 0).

The first experiments concern LH*LH’s performance on the 16-node Parsytec CC platform. Thus, we configured our model to simulate the network topology as shown in Figure 7.4. In the model, the client processes are allocated from nodes 0 up to $c-1$, where c is the number of clients. The servers are placed (when they are created at a split) at the nodes starting from node c. Throughout the remainder of this chapter, we assume that the number of starting servers is one, i.e. $N = 1$. We have also experimented with other mappings of the client and server processes, but we found that the performance differences of the various mappings are negligible.

Because the number of available nodes in the modelled platform is rather limited (16 nodes, to be exact), we needed to calibrate the split-threshold such that the LH*LH SDDS does not grow larger than the available number of nodes. Another result of the limited number of nodes is that the model only simulates up to 5 clients in order to reserve enough nodes for the server part of LH*LH. To overcome these problems, we have also simulated a larger multicomputer platform, of which the results are discussed later in this chapter.

Figure 7.6a shows the prediction of the time it takes to build the LH*LH SDDS when using c clients, where $c = 1, 2, ..., 5$. In the case multiple clients are used, they concurrently insert a different part of the dictionary. The data points in Figure 7.6a correspond to the points in time where LH*LH splits take place. The results show that the build-time scales linearly with the number of insertions. Thus, the insertion latency is independent on the size of the data structure (it is dominated entirely by the message round trip time). As noted earlier, this characteristic is required for a data structure to be scalable. So, in this respect, Figure 7.6a clearly indicates that LH*LH is scalable.

The results of Figure 7.6a also show that the build-time decreases when more clients are used. This is due to the increased parallelism as each client concurrently operates on the LH*LH SDDS. In Figure 7.6b, the relative effect (i.e. the speedup) of the number of

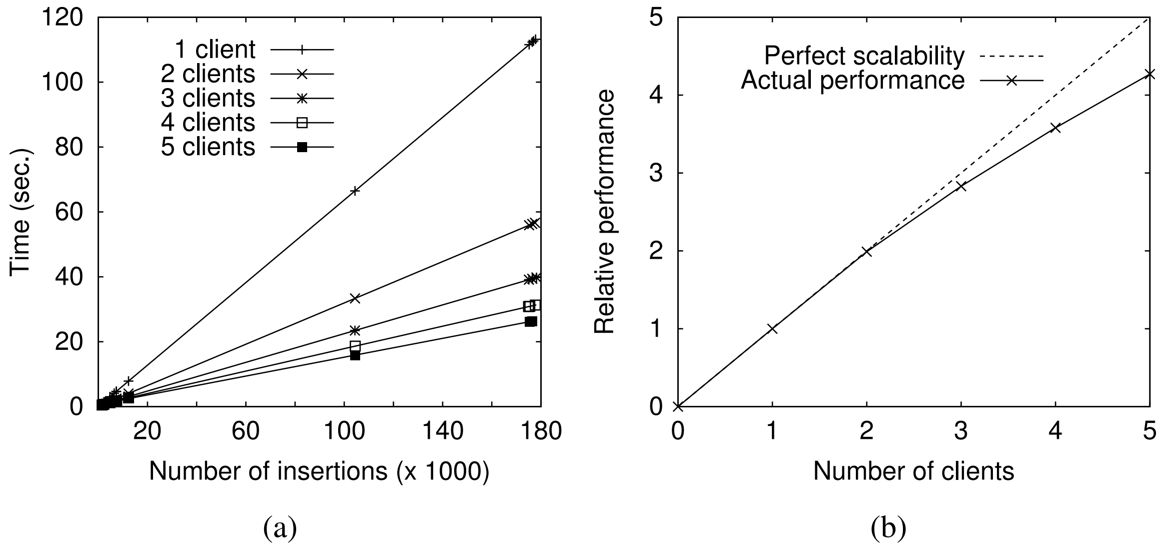


Figure 7.6: Building the LH*LH SDDS: absolute performance (a) and scalability (b). In graph (a), the data points correspond to the moments in time where splits take place.

clients is shown. When increasing the number of clients, the obtained speedup scales quite well for the range of clients used.

Another observation that can be made is that the occurrence of splits (the points in Figure 7.6a) is not uniformly distributed with respect to the insertions. Most of the splits are clustered in the first 20,000 insertions and near the 180,000 insertions. This phenomenon is referred to as *cascading splits* [85]. It is caused by the fact that when the load factor on server n (the token-holder) is high enough to trigger a split, the servers that follow server n are often also ready to split. This means that after server n has split and forwarded the n token to the next server, the new token-holder immediately splits as well. As a result, a cascade of splitting servers is formed which terminates whenever a server is encountered with a load factor that is lower than the threshold. Essentially, cascading splits are undesirable as they harm the incremental fashion with which the distributed data structure is reorganised. In the worst case, the cascade includes all servers which is equal to a total reorganisation of the data structure. Litwin et al. [85] have proposed several adjustments to the split threshold function in order to achieve a more uniform distribution of the splits. Experimental results have shown that these adjustments are quite effective.

Figure 7.7a plots the curves of the average time needed for a single insertion, as experienced by the application. Because the data points for the first 20,000 insertions are relatively hard to distinguish, Figure 7.7b zooms in on this particular range. Figure 7.7a shows that the worst average insertion time (for 1 client) does not exceed 0.65ms. This is about one order of magnitude faster than the typical time to access a disk. However, we should remind the reader that these insertion latencies reflect a lower bound (for the investigated multicomputer architecture) since no computation is modelled at the clients and servers.

As can already be expected from Figure 7.6a, Figure 7.7 shows that the average insertion time decreases when increasing the number of clients. Additionally, the average insertion time also decreases when increasing the number of insertions. This is especially true during the first few thousand of insertions and for the experiments using 3 or more clients. The rea-

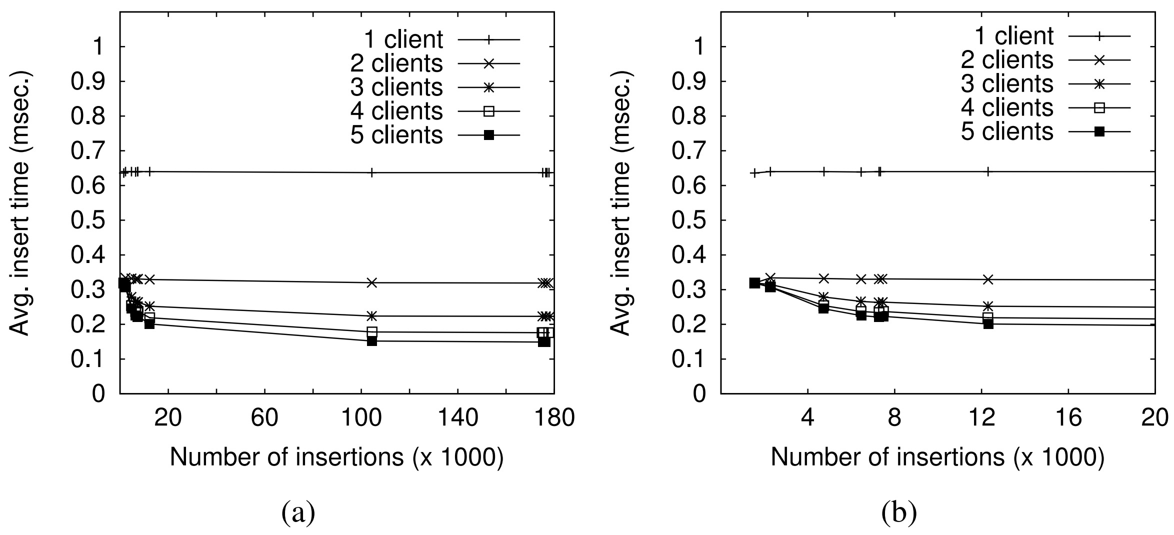


Figure 7.7: Average time per insertion in milliseconds.

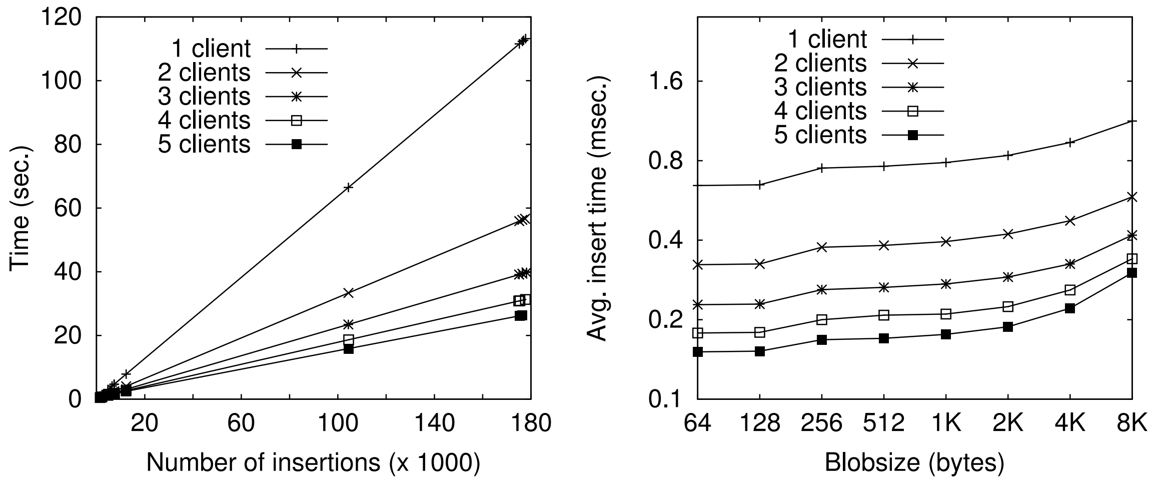
son for this is that a larger number of clients requires the creation of enough servers before the clients are able to effectively exploit parallelism, i.e. allowing the clients to concurrently access the LH*LH SDDS with low server contention. As can be seen in Figure 7.7b, after about 8,000 thousand insertions, enough servers have been created to support 5 clients.

In Table 7.2, the message statistics are shown for building the LH*LH SDDS. For about 180,000 insertions, the maximum number of IAMs (Image Adjustment Messages) that were sent is 55 (for 5 clients). This is only 0.003% with respect to the total number of insertions. Moreover, the average number of messages per insertion is near the optimum of 2 (needed for the request itself and the acknowledgement). The last column of Table 7.2 shows the average message overhead (due to IAMs and the forwarding of messages) for a single insertion. These results confirm the statement from Litwin et al. [85] in which they claimed that it usually takes one message only to address the correct server.

The previously discussed results correspond with the results presented in a study by Karlsson [66, 67]. In this study, an actual implementation of LH*LH was evaluated for a different Parsytec multicomputer, namely the Parsytec GC/PowerPlus. This multicomputer is a predecessor of the Parsytec CC and is based on a mesh topology rather than a multi-stage topology. The resemblance between our simulation results and the LH*LH results for

# Clients	# IAMs	Msg./insertion	Average overhead
1	14	2.0008	0.04%
2	24	2.0011	0.06%
3	43	2.0012	0.06%
4	48	2.0014	0.07%
5	55	2.0014	0.07%

Table 7.2: Number of messages required to build the LH*LH SDDS.



(a) (b)

Figure 7.8: Effect of the blobsize on the performance: the build-time when using a blobsize of 2K (a) and the effect of the blobsize on the average time per insertion (b).

an actual machine can be regarded as a kind of validation; it increases the confidence in our simulation model.

In Figure 7.8, the results are shown when experimenting with the blobsize of data elements. Figure 7.8a plots the curves for the build-time when using a blobsize of 2K (note that the blobsize excludes the key). So, in this experiment the creation of a data structure of nearly 350 Mbytes is simulated. The results show that the build-times are still linear to the number of insertions and that the performance scales properly when adding more clients. In fact, the build-times are not much higher than the ones obtained in the experiment with empty data elements (see Figure 7.6a). The reason for this is twofold. First, since computational latencies like memory references are not simulated, our simulation model is especially optimistic for data elements with a large blobsize. Second, the communication overhead caused by the AIX kernel is rather high (at least 325μs per message) and dominates the communication latency. This reduces the effect of the message size on the communication performance.

Figure 7.8b depicts the average time per insertion when varying the blobsize from 64 bytes to 8K (the latter is building a data structure of 1.4 Gbytes). Note that both axes have a logarithmic scale. Two observations can be made from this figure. First, the average insertion time suddenly increases after a blobsize of 128 bytes. This effect is, however, diminished when using more clients. The reason for this is a peculiarity in the implementation of synchronous communication for the Parsytec CC machine. Messages smaller than 248 bytes can piggy-back on a fast initialisation packet, which sets up the synchronous communication between the source and destination nodes. Beyond these 248 bytes, normal data packets have to be created and transmitted, which slow down the communication (e.g. the software overhead is higher). By increasing the number of clients, and thus increasing the potential parallelism, the larger communication overhead of big messages ($>$ 248 bytes) can be hidden.

A second observation that can be made from Figure 7.8b is that the curves are relatively flat for blobsizes below the 1K. Again, this can be explained by the fact that the large OS

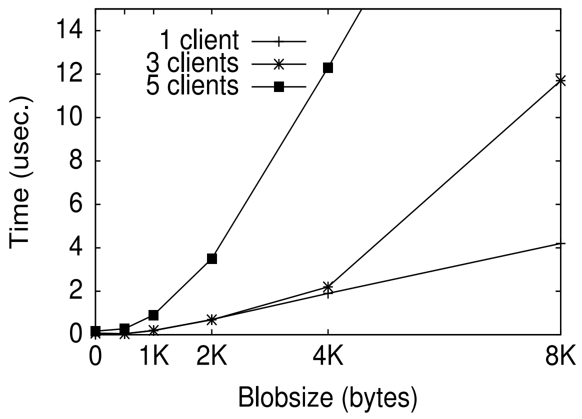
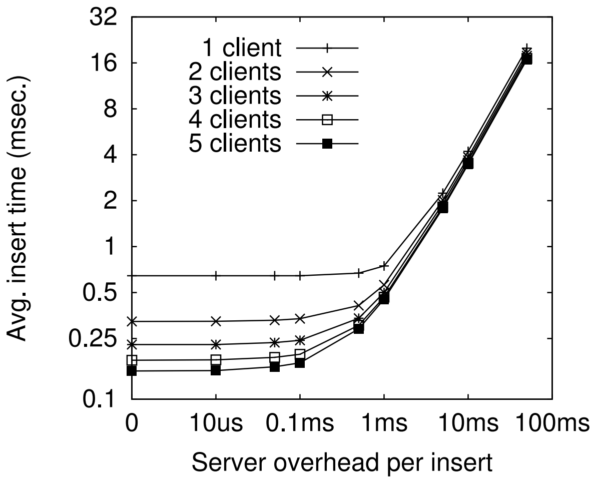


Figure 7.9: The average time that packets blocked within the network.

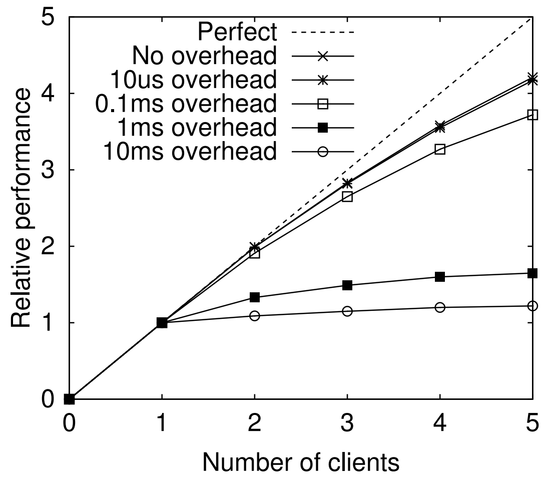
overhead dominates the insertion performance for small blobsizes. For insertions with large blobsizes ($>$ 1K), the network throughput and possibly the network contention become the most dominant performance factor. To investigate whether or not the network contention plays an important role in the insertion performance, Figure 7.9 shows the average time that packets were stalled within the network. We should note that the variation of these averages is rather large, which implies that they should be interpreted with care. Nevertheless, the graph gives a good indication of the intensity of the network contention. From Figure 7.9 can be seen that the contention increases more or less linearly when increasing the blobsize in the case of 1 client. However, when using more clients, the contention starts to scale exponentially. The highest average block time equals to 31μs (8K blobsize with 5 clients), which is only $1/10^{th}$ of the 325μs software communication overhead for a single message. So, in our experiments, the contention is not high enough to dominate the insertion performance. This suggests that the insertion performance for large blobsizes (see Figure 7.8b) is mainly dominated by the network throughput (and not by contention). We return to the topic of contention later in this section, when LH*LH's behaviour is studied for a larger multicomputer platform.

So far, we have assumed that the server (and client) processes are infinitely fast (computation is not modelled). To investigate the effect of server overhead on the overall performance, we modelled a delay for every incoming insertion on a server (the clients continue to be infinitely fast). During the delay, which takes place after acknowledging the client's request, the server is inactive. Since the server overhead can overlap with the transmission of new client requests, this scheme exploits parallelism for all client configurations (even when 1 client is used). We varied the server overhead from 0 (no overhead) to 50ms per insertion. The use of constant overheads is, of course, reasonably simplistic as the overheads may in reality be dependent on issues such as the blobsize, the load factor of the server, etc.

Figure 7.10 depicts the results of the server overhead experiment. In Figure 7.10a, the effect on the average insertion time is shown. Note that both axes have a logarithmic scale. Figure 7.10a shows that the insertion latency starts to be seriously affected by the server overhead after a delay of approximately 0.1ms. Beyond a server overhead of 1ms, the insertion latency increases linearly with the server overhead which indicates that the insertion latency is entirely dominated by the server overhead. After this point, the differences



(a)



(b)

Figure 7.10: Effect of the server overhead on the performance: the average time per insertion (a) and the scalability (b).

between the various client configurations have more or less been disappeared as well, i.e. the curves converge. This implies that the large server overheads reduce the potential parallelism. An important reason for this is that when the server overhead approaches or exceeds the minimum time between two consecutive insertion requests from a single client (which is in our case simply the 325μs communication latency as the clients do not perform any computation), the rate at which a server can process insertion requests becomes lower than the rate at which a single client can produce requests. Evidently, this means that a server can easily become a bottleneck when adding clients.

The reduction of parallelism caused by large server overheads is best illustrated in Figure 7.10b. This figure shows the speedup for multiple clients when varying the server overhead. It is obvious that while the scalability is good for server overheads of 0.1ms and less, the scalability for larger overheads has collapsed completely.

Until now, the experiments have been performed using the simulation model of a 16-node Parsytec CC architecture. This has limited our simulations to allocating a maximum of 5 clients and 11 servers. To investigate how LH*LH behaves for a larger number of clients and servers, we have also experimented with larger networks. More specifically, we have modelled two 64-node networks: a Mesh of Clos(3,1) (see Figure 5.4) and an 8×8 mesh. We found, however, that the results are almost identical for these two 64-node networks (the differences are less than 2%). This is readily explained by the fact that our LH*LH workload does not generate the network contention which is necessary to notice performance differences between the two networks. Therefore, we will only present the results for the Mesh of Clos(3,1) network.

In the Mesh of Clos model, we have used a different allocation scheme for the client processes as compared to one from the Parsytec CC model. This is because the latter scheme (allocating nodes 0 to $c - 1$ for c client processes) would centralise the clients at one or two clusters within the Mesh of Clos(3,1) network. As a consequence, most communication needs to cross the mesh part of Mesh of Clos, which is more prone to contention (see Chapter 5). Therefore, we have distributed the clients evenly over the four Clos clusters of

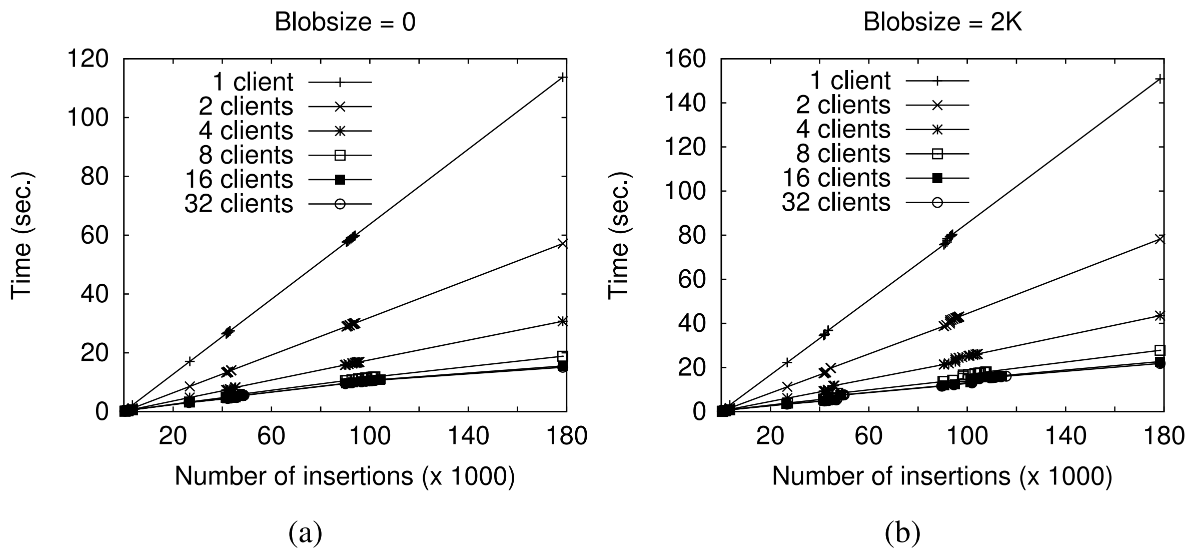


Figure 7.11: Build-times for the MoC(3,1) network using blobsizes of 0 (a) and 2K (b).

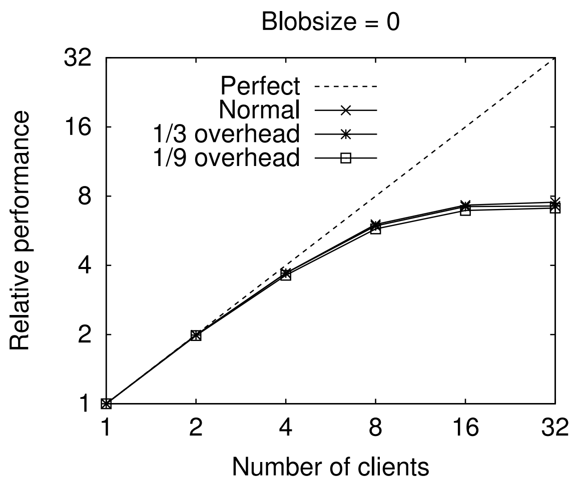
the Mesh of Clos. Unfortunately, the performance improvements due to the new allocation scheme are quite insignificant (a few percents only). We suspect that this is again due to the relative low network contention. Because the performance improvements are marginal, we do not show them in this case study.

Figure 7.11 shows the build-times for the Mesh of Clos(3,1) when using blobsizes of 0 (Figure 7.11a) and 2K (Figure 7.11b). Again, the data points refer to the moments in time where a split occurs. We adapted LH*LH's split-threshold such that the distributed data structure grows to nearly 32 servers. As a result, we were able to simulate up to 32 clients. The curves in Figure 7.11 show the same behaviour as was observed in the experiments with the Parsytec CC model: the build-time is linear to the number of insertions and decreases with an increasing number of clients. The occurrence of cascading splits is also illustrated by the clusters of points in Figure 7.11.

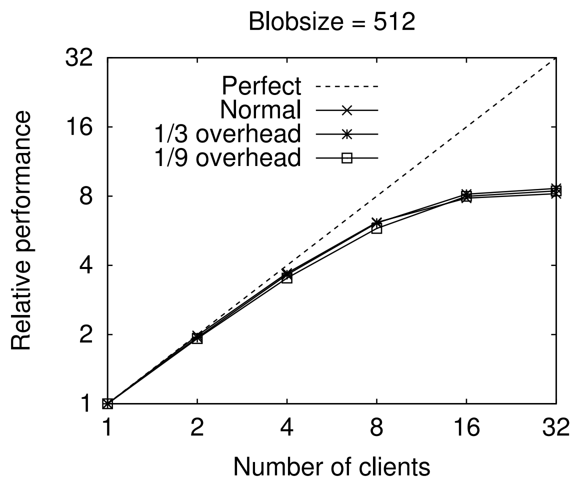
In Figure 7.12, the scalability for several blobsizes is plotted by the curves which are labelled with *normal*. Additionally, the curves labelled with "1/3" and "1/9" show the scalability when the OS communication overhead is reduced by a factor 3 and 9 respectively. Remember that the original overhead (the *normal* curve) equals to roughly 325μs per message.

A number of observations can be made from Figure 7.12. First, the *normal* curves indicate that the configurations with blobsizes up to 2K scale to roughly 8 clients, whereas the configuration with a blobsize of 8K only scales up to about 4 clients. The deterioration of scalability for large blobsizes is due to the increased network contention. Remember Figure 7.9, in which it was shown that the contention already grows exponentially with the blobsize when using five clients.

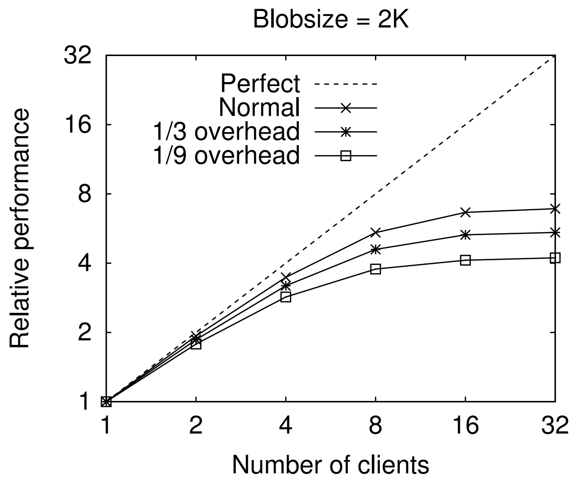
Another, quite interesting, result is that the scalability for the large blobsizes is reduced even more when decreasing the software communication overhead. This effect is caused by an increase of network contention: the smaller the overhead, the higher the frequency at which the clients inject insertion messages into the network. Clearly, the higher frequency of insertions with large blobsizes results in more network traffic and thus more contention.



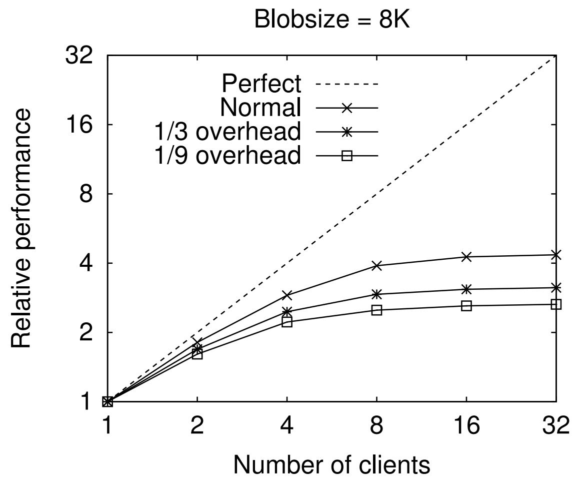
(a)



(b)



(c)



(d)

Figure 7.12: The scalability for blobsizes of 0 bytes (a), 512 bytes (b), 2K (c) and 8K (d). The curve labelled with *normal* refers to the results of the original model. The other two curves (1/3 and 1/9) present the results for a modified model in which the software communication overhead is reduced by a factor of 3 and 9 respectively.

So, one can conclude that a part of the measured scalability for large blobsizes is due to the high communication overhead of the modelled platform. Apparently, the increase of network traffic in the case of a blobsize of 512 bytes is not severe enough to cause a lot of extra contention.

7.5 Discussion

In this case study, we have shown how Mermaid can be used for evaluating the performance at the application level. For this purpose, we evaluated the LH*LH distributed data structure for a multicomputer architecture. This data structure, which is based on linear hashing, should provide fast access to a vast amount of (distributed) data. To do so, the data structure is designed to be scalable, which can be characterised by three requirements: (i) operations on the distributed data are not dependent on the amount of data stored, (ii) the data structure can handle any amount of data and (iii) the data structure grows and shrinks incrementally over the distributed resources, i.e. the servers.

We studied LH*LH's scalability behaviour with a benchmark that builds the distributed data structure using a dictionary for the insertion keys. In this evaluation, we were mainly interested in the communication load that is generated by LH*LH. Therefore, we have used an adapted version of the communication model from Chapter 5 to perform the architectural simulation.

Our simulation results confirm that LH*LH is indeed scalable (according to the previously mentioned requirements). We found that the time to insert a data element is independent on the size of the data structure. For the studied multicomputer, the insertion time as experienced by a single client can be an order of magnitude faster than a typical disk access. The insertion time can even be reduced by using multiple clients which concurrently insert data into the distributed data structure. The results indicate that the speedup of insertions scales reasonably well up to about 8 clients for the investigated network architecture.

We have also shown that the scalability of insertion performance can be affected in more than one way. For instance, the larger the data elements that are inserted, the poorer is the scalability when increasing the number of clients. We found, for example, that the performance of insertions with data elements of 512 bytes scales up to 8 clients, whereas the performance of insertions with 8K data elements only scales up to 4 clients. Moreover, a large server overhead (we experimented with constant computational overheads) can seriously hamper the client-scalability of the LH*LH data structure. Our results indicate that it is important to keep the server overhead below the minimum time between two consecutive requests from a single client. This increases the potential for exploiting parallelism as it ensures that the rate at which a server can process requests is higher than the rate at which a single client can produce requests. Finally, we found that a part of the speedup for multiple clients is due to the hiding of software communication overhead (which is quite large for the studied architecture). When the communication is optimised (i.e. the overhead is reduced), lower speedups are achieved.

For several experiments, we were able to compare our simulation results with the results from a real LH*LH implementation. Although the measurements for the real implementation were obtained using a multicomputer which is different from the one we have modelled, the comparison between our simulation results and the actual execution results can still give us some insight into the validity of our simulation model. We found that the simulation results correspond closely to the behaviour measured for the real implementation.

Throughout this study, we focused on the evaluation of the application level, being the LH*LH data structure. In the future, we might extend these experiments to include more aspects of LH*LH's functionality (e.g. the way in which buckets are shipped, different types of split-threshold functions, etc.). Alternatively, we could also focus on the architecture level. At this level, aspects such as the influence of the routing scheme or the type of parallel platform (e.g. using a cluster of workstations rather than a multicomputer) can be investigated for the LH*LH workload.

Chapter 8

Conclusions

"Our achievements speak for themselves. What we have to keep track of are our failures, discouragements, and doubts. We tend to forget the past difficulties, the many false starts, and the painful groping."

Eric Hoffer

Performance evaluation by means of simulation plays an important role in the design cycle of computer architectures. It allows for accurately exploring the design space, which is essential for optimising the computer architecture's speed. In the second chapter of this thesis, we have presented an overview of the simulation techniques that are commonly used when evaluating the performance of computer architectures. We ended the overview by focusing on the simulation methods that are suitable for the study of parallel computers. Here, we concluded that most of the existing simulators apply techniques that trade simulation speed for flexibility. For example, several of these simulation methods assume that the instruction sets of the host computer (on which the simulation is executed) and the destination architecture (which is simulated) are identical. Such architectural dependencies are, in our opinion, undesirable. They affect the ease with which the simulation models are constructed or, even worse, hamper the modelling freedom. The need for modelling flexibility is well illustrated by the diversity of the three case studies presented in the last chapters of this thesis.

In this thesis, we have addressed the tradeoff between simulation performance and flexibility by introducing a new simulation methodology, called operation-driven simulation. This methodology is a combination of traditional trace-driven simulation and execution-driven simulation. The trace events, which are called operations, can be chosen such that their abstraction level fits the architecture specifics in which we are interested. For example, the operations can be (abstract) machine instructions in order to study the performance behaviour with relative high accuracy or they can specify computation and communication at a much higher level (e.g. at task level) to establish a fast but less accurate type of simulation which might be useful for the purpose of fast-prototyping. So, in other words, we use abstraction as an instrument to control the speed and, as a consequence, the accuracy of the simulation.

Operation-driven simulation allows for a high degree of modelling flexibility. The reason for this is twofold. First, the operations can be chosen such that they are not dependent on the host's nor the destination's architecture. This implies that, for example, different processor types (with non-identical instruction sets) can be simulated without changing the simulator. Second, the operation-based interface of the simulator allows for decoupling the modelling of application behaviour (i.e. the generation of operations) and architectural behaviour. More specifically, the generation of the operation events can now be performed at various levels of abstraction and with different degrees of accuracy. For instance, the behaviour of a real program can be traced to obtain a realistic reflection of application behaviour or some stochastic process can generate the operations. The latter technique is flexible (application behaviour is easily changed) but not highly accurate.

In Chapter 3, the Mermaid simulation environment was presented in which we have embedded the operation-driven simulation methodology. This environment provides a workbench for the performance evaluation of parallel computer architectures and, in particular, multicomputer architectures. Chapter 4 has shown that Mermaid's flexibility, obtained by the operation-driven simulation engine, does not compromise the efficiency and accuracy of the simulation. Validation experiments have demonstrated that, despite the high abstraction level of our simulation models, accurate performance predictions are achieved. Also, we have shown that Mermaid's simulation performance is competitive with other, state-of-the-art parallel architecture simulators. To boost the simulation performance even more, we have added functionality to Mermaid which allows for distributed execution of the simulation on a cluster of workstations. The resulting distributed simulator outperforms nearly all other parallel architecture simulators.

Chapters 5 to 7 present three case studies in which Mermaid has been applied. In the first study, a wormhole-routed network with a Mesh of Clos topology is evaluated using a set of synthetic communication workloads. The second study zooms in on the performance behaviour of a single processor only. In this study, we evaluated several prefetching techniques for the data cache of a VLIW processor. Finally, the third case study investigates the performance behaviour of a scalable, distributed data structure for a multicomputer architecture.

The diversity of the case studies together with the fact that we were able to readily use Mermaid for all three of them clearly illustrates the flexibility of our simulation methodology. It also shows that Mermaid is capable of evaluating the performance at both the architecture level (Chapters 5 and 6) and the application level (Chapter 7).

For two case studies, we were able to compare some of our simulation results with the results from either a real machine or a highly detailed simulator. These validation experiments confirm what was already found in Chapter 4, namely that a reasonably abstract model can obtain good simulation accuracy.

8.1 Future work

This thesis has described research that is ongoing. There is still room for improvement, especially at the application level of our simulation environment. For example, we would like to investigate techniques to generate *and validate* realistic, stochastic multicomputer traces. In particular, the validation of stochastic traces is non-trivial. For instance, the locality of reference is hard to quantify as there is still no clear metric for locality.

Furthermore, we should make our reality-based workload modelling more robust. Currently, Mermaid only allows for instrumenting and tracing C programs. The capability of handling alternative languages, such as Fortran or Java, would take Mermaid to an even higher level of flexibility.

Bibliography

[1] A. Agarwal, M. Horowitz, and J. L. Hennessy. An analytic cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, May 1989.

[2] A. Agarwal, R. L. Sites, and M. Horowitz. ATUM: A new technique for capturing address traces using microcode. In *Proc. of the 13th Int. Symposium on Computer Architecture*, pages 119–127, 1986.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Massachusetts, 1986.

[4] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the 1990 ACM Int. Conference on Supercomputing*, pages 1–6, 1990.

[5] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Spring Joint Computer Conf.*, number 30 in AFIPS conf. proc., pages 483–485. Thompson Books, Academic Press, April 1967.

[6] P. America and J. Rutten. *A parallel object-orientated language: design and semantic foundations*. PhD thesis, Free University, Amsterdam, May 1989.

[7] D. Badouel, C. A. Wüthrich, and E. L. Fiume. Routing strategies and message contention on low-dimensional interconnection networks. Technical report, Comp. System Research Institute, University of Toronto, Dec 1991.

[8] R. C. Bedichek. *The Meerkat Multicomputer: Trade-offs in Multicomputer Design*. PhD thesis, Dept. of Computer Science, University of Washington, Aug. 1994.

[9] R. C. Bedichek. Talisman: Fast and accurate multicomputer simulation. In *Proceedings of the 1995 ACM SIGMETRICS Conference*, pages 14–24, May 1995.

[10] M. Beemster. *Fine-grained parallelism in a lazy functional language*. PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, March 1996.

[11] A. D. Birrell. An introduction to programming with threads. Technical Report 35, Digital Systems Research Center, Jan. 1989.

[12] G. M. Birtwistle, O. J. Dahl, B. Myhrhaug, and K. Nygaard. Simula begin, 1973.

[13] B. Black and J. P. Shen. Rigorous validation of superscalar performance models. In *Proc. of the Workshop on Performance Analysis and its Impact on Design (in conjunction with the 24th Int. Symposium on Computer Architecture)*, pages 64–70, June 1997.

[14] M. T. Bohr. Interconnect scaling – the real limiter to high performance ULSI. In *Int. Electron Devices Meeting Technical Digest*, pages 241–244, 1995.

[15] P. A. Boncz and M. L. Kersten. Monet: An impressionist sketch of an advanced database system. In *Proc. of IEEE BIWIT workshop*, July 1995.

[16] B. Boothe. Fast accurate simulation of large shared memory multiprocessors. Technical Report CSD 92/682, Comp. Science Div. (EECS), Univ. of California at Berkeley, June 1993.

[17] A. Borg, R. Kessler, and D. Wall. Generation and analysis of very long address traces. In *Proc. of the 17th Int. Symposium on Computer Architecture*, pages 270–281, 1990.

[18] E. A. Brewer, Ch. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, Sept. 1991.

[19] R. B. Bunt and J. M. Murphy. The measurement of locality and behaviour of programs. *The Computer Journal*, 27(3):238–245, 1984.

[20] F. E. Cellier. *Continuous System Modeling*. Springer-Verlag, 1991.

[21] T-F. Chen. An effective programmable prefetch engine for on-chip caches. In *Proc. of the 28th Int. Symposium on Microarchitecture*, pages 237–242, Nov. 1995.

[22] T-F. Chen and J-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[23] S. Chodnekar, V. Srinivasan, A. Vaidya, A. Sivasubramaniam, and C. Das. Towards a communication characterization methodology for parallel applications. In *Proc. of the 3rd Int. Symposium on High Performance Computer Architecture (HPCA)*, pages 310–319, Feb. 1997.

[24] C. Clos. A study of non blocking switching networks. *Bell System Technical Journal*, pages 775–785, Mar. 1953.

[25] R. F. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proc. of the 1994 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 128–137, May 1994.

[26] IEEE Computer. Special issue on future microprocessors. 30(9), Sept. 1997.

[27] T. M. Conte and C. E. Gimarc, editors. *Fast Simulation of Computer Architectures*. Kluwer Academic Publishers, 1995.

[28] R. G. Covington, S. Dwarkadas, J. R. Jump, J. B. Sinclair, and S. Madala. The efficient simulation of parallel computer systems. *Int. Journal in Comp. Simulation*, 1:31–58, 1991.

[29] R. G. Covington and J. R. Jump. Csim 2.0 users guide. Technical Report TR8501, Elec. and Comp. Eng. Dept., Rice University, Feb. 1986.

[30] W. J. Dally. Virtual channel flow control. In *Proc. of the 17th Int. Symposium on Computer Architecture*, pages 60–68, May 1990.

[31] W. J. Dally and C. L. Seitz. The torus routing chip. *Journal of Distributed Computing*, 1(3):187–196, 1986.

[32] P. Davies, P. Lacroute, J. Heinlein, and M. Horowitz. Mable: a technique for efficient machine simulation. Technical Report CSL-TR-94-636, Stanford University, Oct. 1994.

[33] H. Davis, S. R. Goldschmidt, and J. Hennessy. Multiprocessor simulation and tracing using Tango. In *Proc. of the 1991 Int. Conf. in Parallel Processing*, pages 99–107, Aug. 1991.

[34] M. Dubois, F. A. Briggs, I. Patil, and M. Balakrishnan. Trace-driven simulations of parallel and distributed algorithms in multiprocessors. In *Proc. of the 1986 Int. Conference in Parallel Processing*, pages 909–915, Aug. 1986.

[35] S. Eggers, D. Keppel, E. Koldinger, and H. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *Proc. of the 1990 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 37–47, 1990.

[36] G. H. Barnes et al. The ILLIAC IV computer. *IEEE Transactions*, C-17:746–757, 1968.

[37] S. A. Felperin, L. Gravano, G. D. Pifarre, and J. L. Sanz. Routing techniques for massively parallel communication. In *Proceedings of the IEEE*, volume 79, pages 488–503, Apr. 1991.

[38] J. K. Flanagan, B. E. Nelson, J. K. Archibald, and K. Grimsrud. BACH: BYU address collection hardware, the collection of complete traces. In *Proc. of the 6th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 128–137, 1992.

[39] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept 1972.

[40] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics, Principles and Practice, Second Edition.* Addison-Wesley, Reading, Massachusetts, 1990.

[41] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, 1994.

[42] J. W. C. Fu and J. H. Patel. Stride directed prefetching in scalar processors. In *Proc. of the 25th Int. Symposium on Microarchitecture*, pages 102–110, 1992.

[43] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[44] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith. Cache performance of the SPEC92 benchmark suite. *IEEE Micro*, 13(4):17–27, Aug. 1993.

[45] S. R. Goldschmidt. *Simulation of Multiprocessors: Accuracy and Performance*. PhD thesis, Dept. of Electrical Eng., Stanford University, June 1993.

[46] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *Proc. of the 1993 ACM SIGMETRICS Conference*, pages 146–157, May 1993.

[47] TriMedia Product Group. *TM-1 Preliminary Data Book*, March 1997.

[48] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W-D. Weber. Comparative evaluation of latency reducing and tolerating techniques. In *The 18th Int. Symposium on Computer Architecture*, pages 254–263, May 1991.

[49] R. Gupta. SPMD execution of programs with dynamic data structures on distributed memory machines. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 232–241, Apr. 1992.

[50] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(3):532–533, May 1988.

[51] L. Gwennap. Intel, HP make EPIC disclosure. In *Microprocessor Report*, Oct. 1997.

[52] J. L. Hennessy and D. A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[53] M. A. Holliday and C. S. Ellis. Accuracy of memory reference traces of parallel computations in trace-driven simulation. *IEEE Transactions on Parallel and Distributed Systems*, 3(1):97–109, Jan. 1992.

[54] HPF-Forum. High Performance Fortran Language Specification, version 1.0. *Scientific Programming*, 2(1-2):1–170, 1993.

[55] J.-M. Hsu and P. Banerjee. Performance measurement and trace driven simulation of parallel CAD and numeric applications on a hypercube multicomputer. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):451–464, July 1992.

[56] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August. Compiler technology for future microprocessors. *Proceedings of the IEEE*, 83(12):1625–1640, Dec. 1995.

[57] Projectvoorstel: HPCN in de Financiele Diensten Sector. Groot-schalige parallelle gegevensverwerking en applicatie ontwikkeling, March 1995.

[58] Inmos. *The Transputer Databook.* Inmos Ltd., 1992.

[59] R. Jain. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, Inc., 1991.

[60] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985. D:3.3.

[61] E. E. Johnson and J. Ha. PDATS: Lossless address trace compression for reducing file size and access time. In *Proc. of the IEEE Int. Phoenix Conf. on Computers and Communications*, pages 213–219, 1994.

[62] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *24th Int. Symposium on Computer Architecture*, pages 252–263, June 1997.

[63] A. G. P. Joubert. SPAM: A multiprocessor execution driven simulation kernel. Technical Report 708, IRISA research laboratory, Mar. 1993.

[64] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *The 17th Int. Symposium on Computer Architecture*, pages 364–373, May 1990.

[65] T. A. Jung, R. R. Schlittler, J. K. Gimzewski, H. Tang, and C. Joachim. Controlled room-temperature positioning of individual molecules: Molecular flexure and motion. *Science*, 271(5246), Jan. 1996.

[66] J. S. Karlsson. A scalable data structure for a parallel data server. Master's thesis, Dept. of Comp. and Inf. Science, Linköping University, Feb. 1997.

[67] J. S. Karlsson, W. Litwin, and T. Risch. LH*lh: A scalable high performance data structure for switched multicomputers. In *Advances in Database Technology — EDBT '96*, pages 573–591, March 1996.

[68] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3(4):267–286, 1979.

[69] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1978.

[70] H. C. Kok. Visualizing computer architecture simulations — graphical user interface support for Pearl. Master's thesis, Dept. of Comp. Sys, Univ. of Amsterdam, Aug. 1996.

[71] H. C. Kok, A. D. Pimentel, and L. O. Hertzberger. Runtime visualization of computer architecture simulations. In *Proc. of the Workshop on Performance Analysis and its Impact on Design (in conjunction with the 24th Int. Symposium on Computer Architecture)*, pages 15–24, June 1997.

[72] KSR. KSR technical summary, 1992.

[73] H.T. Kung. Why systolic architectures? *IEEE Computer*, 15(1):37–46, 1982.

[74] S. Laha, J. Patel, and R. Lyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, 37(11):1325–1336, 1988.

[75] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communication of the ACM*, 21(7):558–565, July 1978.

[76] J. R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software Practice & Experience*, 20(12):1241–1258, Dec. 1990.

[77] J. R. Larus. SPIM S20: A MIPS R2000 simulator. Technical Report , Revision 9, University of Wisconsin-Madison, 1991.

[78] J. R. Larus. Efficient program tracing. *IEEE Computer*, pages 52–60, May 1993.

[79] J. R. Larus and T. Ball. Rewriting executable files to measure program behaviour. *Software Practice & Experience*, 24(2):197–218, Feb. 1994.

[80] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proc. of the 24th Int. Symposium on Computer Architecture*, pages 241–251, June 1997.

[81] E. Lazowska, J. Zahorja, G. Graham, and K. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, 1984.

[82] A. R. Lebeck and D. A. Wood. Active memory: A new abstraction for memory system simulation. *ACM Transactions on Modeling and Simulation*, 7(1):42–77, Jan. 1997.

[83] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proc. of VLDB*, 1980.

[84] W. Litwin. Linear hashing: A new tool for file and table addressing. In M. Stonebraker, editor, *Readings in DATABASE SYSTEMS*, pages 96–107. Morgan Kaufmann, 1995.

[85] W. Litwin, M-A. Neimat, and D. Schneider. LH*: A scalable, distributed data structure. *ACM Transactions on Database Systems*, 21(4):480–526, Dec. 1996.

[86] T. Lovett and R. Clapp. STiNG: A ccNUMA compute system for the commercial marketplace. In *Proc. of the 23rd Int. Symposium on Computer Architecture*, pages 308–317, May 1996.

[87] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proc. of 25th International Symposium on Microarchitecture*, pages 45–54, Dec. 1992.

[88] P. K. McKinley and C. Trefftz. MultiSim: A simulation tool for the study of large-scale multiprocessors. In *Proceedings of the 1993 International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Networks (MASCOTS)*, pages 57–62, Jan. 1993.

[89] F. H. McMahon. Llnl fortran kernels: Mflops. Technical report, Lawrence Livermore Laboratories, CA, March 1984.

[90] L. F. Menabrea. A sketch of the Analytical Engine invented by Charles Babbage, Bibiotheque Universelle de Geneve, Oct. 1842.

[91] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[92] B. Monien, R. Lüling, and F. Langhammer. A realizable efficient parallel architecture. In *Proc. of the 1st Int. Heinz Nixdorf Symposium, LNCS*, volume 678, pages 93–109, 1992.

[93] W. G. P. Mooij. *Packet Switched Communication Networks for Multi-Processor Systems*. PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, Dec 1989.

[94] Motorola. *PowerPC 601 RISC Microprocessor User's Manual*. Motorola Inc., 1993.

[95] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[96] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood. Wisconsin Wind Tunnel II: A fast and portable parallel architecture simulator. In *Proc. of the Workshop on Performance Analysis and its Impact on Design (in conjunction with the 24th Int. Symposium on Computer Architecture)*, June 1997.

[97] H. L. Muller. *Simulating computer architectures*. PhD thesis, Dept. of Comp. Sys, Univ. of Amsterdam, Feb. 1993.

[98] H. L. Muller, K. G. Langendoen, and L. O. Hertzberger. MiG: Simulating parallel functional programs on hierarchical cache architectures. Technical Report CS-92-04, Dept. of Comp. Sys, Univ. of Amsterdam, June 1992.

[99] SPEC Newsletter, 3(4), 1991.

[100] L. M. Ni and P. K. McKinley. A survey of routing techniques in wormhole networks. Technical Report MSU-CPS-ACS-46, Dept. of Comp. Sc., Michigan State University, Oct. 1991.

[101] L. M. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, 26:62–76, Feb. 1993.

[102] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proc. of the 24th Int. Symposium on Computer Architecture*, pages 206–218, June 1997.

[103] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proc. of the 21th Int. Symposium on Computer Architecture*, pages 24–33, April 1994.

[104] F. Petrini. Personal communication, Feb. 1998.

[105] F. Petrini and M. Vanneschi. SMART: a Simulator of Massive ARchitectures and Topologies. In *Proceedings of the Int. Conference on Parallel and Distributed Systems Euro-PDS'97*, June 1997.

[106] G. Pfister and V. Norton. Hot spot contention and combining in multistage interconnection netw orks. *IEEE Transactions on Computers*, 34(10):943–948, Oct. 1985.

[107] A. D. Pimentel and L. O. Hertzberger. Evaluation of a Mesh of Clos wormhole network. In *Proc. of the 3rd Int. Conference on High Performance Computing*, pages 158–164. IEEE Computer Society Press, Dec. 1996.

[108] A. D. Pimentel and L. O. Hertzberger. Abstract workload modelling in computer architecture simulation. In *Proc. of the Workshop on Performance Analysis and its Impact on Design (in conjunction with the 24th Int. Symposium on Computer Architecture)*, pages 6–14, June 1997.

[109] A. D. Pimentel and L. O. Hertzberger. An architecture workbench for multicomputers. In *Proc. of the 11th Int. Parallel Processing Symposium*, pages 94–99. IEEE Computer Society Press, April 1997.

[110] A. D. Pimentel and L. O. Hertzberger. RAPID: RAPid Interpretation of Data. Technical Report CS-97-01, Dept. of Comp. Sys, Univ. of Amsterdam, Jan. 1997.

[111] A. D. Pimentel and L. O. Hertzberger. Distributed simulation of multicomputer architectures with Mermaid. In *Proc. of the SCS Symposium on Performance Evaluation of Computer and Telecommunication Systems*, pages 73–79, July 1998.

[112] A. D. Pimentel, P. Struik, and P. van der Wolf. Evaluation of hardware-based and hybrid prefetching techniques for the TM1. Technical Report PROMMPT-175, Philips Research Laboratories, Nov. 1997.

[113] A. D. Pimentel, P. Struik, and P. van der Wolf. Scalar versus stream data prefetching for the TriMedia. Technical Report PROMMPT-191, Philips Research Laboratories, March 1998.

[114] A. D. Pimentel, J. van Brummen, Th. Papathanassiadis, P. M. A. Sloot, and L. O. Hertzberger. Mermaid: Modelling and Evaluation Research in MIMD ArchItecture Design. In *Proc. of the High Performance Computing and Networking Conference, LNCS*, pages 335–340, May 1995.

[115] L. F. Pollacia. A survey of discrete event simulation and state-of-the-art discrete event languages. *Simulation Digest*, 20(3):8–25, 1989.

[116] T. Puzak. *Analysis of cache replacement algorithms*. PhD thesis, University of Massachusetts, 1985.

[117] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proc. of the 1993 ACM SIGMETRICS Conference*, pages 48–60, May 1993.

[118] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modelling and Computer Simulation*, 7(1):78–103, Jan. 1997.

[119] M. Rosenblum, S. A. Herrod, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel & Distributed Technology*, 3(4):34–43, 1995.

[120] M. Röttger, U. Schroeder, and J. Simon. Virtual topology library for PARIX. Technical Report TR-005-93, Paderborn Center for Parallel Computing (PC^2), Dept. of Mathematics and Computer Science, University of Paderborn, Germany, 1993.

[121] A. D. Samples. Mach: No-loss trace compaction. In *Proc. of the 1989 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 89–97, May 1989.

[122] V. Santhanam, E. H. Gornish, and W. Hsu. Data prefetching on the HP PA-8000. In *24th Int. Symposium on Computer Architecture*, pages 264–273, June 1997.

[123] C. L. Seitz. The cosmic cube. *Communications of the ACM*, 28:22–33, Jan. 1985.

[124] F. Sijstermans, E. Pol, B. Riemens, K. Vissers, S. Rathnam, and G. A. Slavenburg. Design space exploration for future TriMedia CPUs. In *Proc. of the 23rd Int. Conf. on Acoustics, Speech and Signal Processing*, May 1998.

[125] D. Sima, T. Fountain, and P. Kacsuk. *Advanced Computer Architecture, A Design Space Approach*. Addison Wesley, 1997.

[126] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran. A simulation-based scalability study of parallel systems. *Journal of Parallel and Distributed Computing*, 22:411–426, 1994.

[127] G. A. Slavenburg, S. Rathnam, and H. Dijkstra. The TriMedia TM-1 PCI VLIW media processor. In *Proc. of Hot Chips 8*, pages 171–177, Aug. 1996.

[128] P. M. A. Sloot. Modelling and simulation. In *Proceedings of the 1994 CERN School of Computing*, pages 177–226, Sopron, Hungary, Sept. 1994.

[129] P. M. A. Sloot, A. D. Pimentel, and L. O. Hertzberger. Design issues for high performance simulation. *Simulation Practice and Theory*, 6(3):221–242, 1998.

[130] A. J. Smith. Two methods for the efficient analysis of memory address trace data. *IEEE Transactions on Software Engineering*, 3(1):94–101, 1977.

[131] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[132] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proc. of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[133] P. Stenström. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, pages 12–24, June 1990.

[134] P. Struik, P. van der Wolf, and A. D. Pimentel. A combined hardware/software solution for stream prefetching in multimedia applications. In *Proc. of the 10th Annual Symposium on Electronic Imaging*, pages 120–130, Jan. 1998.

[135] C. B. Stunkel, B. Janssens, and W. K. Fuchs. Address tracing for parallel machines. *IEEE Computer*, 24(1):31–38, Jan. 1991.

[136] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[137] R. Uhlig, D. Nagle, T. Mudge, and S. Sechrest. Trap-driven memory simulation with Tapeworm II. *ACM Transactions on Modeling and Simulation*, 7(1):7–41, Jan. 1997.

[138] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2):128–170, 1997.

[139] C. A. J. van Eijk. *Formal Methods for the Verification of Digital Circuits*. PhD thesis, Eindhoven University of Technology, Sept. 1997.

[140] A. J. C. van Gemund. *Performance Modeling of Parallel Systems*. PhD thesis, Delft University of Technology, Apr. 1996.

[141] D. H. D. Warren and S. Haridi. The Data Diffusion Machine – a scalable shared virtual memory multiprocessor. In *Proc. of 1988 Int. Conference on Fifth Generation Computer Systems*, pages 943–952, Dec. 1988.

[142] M. S. Warren, J. K. Salmon, D. J. Becker, M. P. Goda, T. Sterling, and G. S. Winckelmans. Pentium Pro inside: I. a treecode at 430 gigaflops on ASCI Red, II. price/performance $50/Mflop on Loki and Hyglac. In *Proc. of Supercomputing '97*, Nov. 1997.

[143] K. Whang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, Inc., 1993.

[144] B. P. Zeigler. *Multifaceted Modeling and Discrete Event Simulation*. Academic Press, London, 1984.

[145] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:75–81, 1967.

[146] D. F. Zucker, M. J. Flynn, and R. B. Lee. A comparison of hardware prefetching techniques for multimedia benchmarks. In *Proc. of Int. Conference on Multimedia Computing and Systems*, pages 236–244, June 1996.

[147] D. F. Zucker, R. B. Lee, and M. J. Flynn. An automated method for software controlled cache prefetching. In *Proc. of the Thirty-first Hawaii International Conference on System Sciences*, Jan. 1998.

Dutch summary / Nederlandse samenvatting

Het ontwerpen van computer architecturen is een moeilijke taak door het grote aantal keuzes en ontwerp-afwegingen waarmee een architect te maken heeft. Daarom vormt de prestatie analyse door middel van simulatie een essentieel onderdeel van de ontwerp cyclus. Het in software simuleren van een computer architectuur stelt de architect in staat om verschillende ontwerp-afwegingen nauwkeurig te evalueren en zodoende de prestatie van het uiteindelijke ontwerp te optimaliseren.

Dit proefschrift handelt over de simulatie-gebaseerde prestatie analyse van parallelle computer architecturen. Deze parallelle computers bevatten meerdere processoren waarover het rekenwerk verdeeld kan worden, met als doel om de totale rekentijd van een programma te reduceren. Het onderzoek richt zich specifiek op machines met gedistribueerd geheugen, oftewel de zogenaamde *multicomputers*. De processoren in deze multicomputers bezitten allemaal een eigen geheugen en communiceren met elkaar door middel van het versturen van boodschappen.

In hoofdstuk 2 van dit proefschrift wordt de lezer bekend gemaakt met een aantal algemene simulatie methoden en technieken die veel gebruikt worden in de prestatie analyse van computer architecturen. Aan het einde van dit hoofdstuk worden de technieken besproken die specifiek geschikt zijn voor het simuleren van parallelle computers. Hier concludeerde wij dat de huidige parallelle architectuur simulatoren vaak flexibiliteit inleveren om hogere simulatie snelheden te bereiken. Zo gaan bijvoorbeeld sommige simulatie methoden er van uit dat de instructie sets van de gastheer computer (waarop de simulatie executeert) en van de te evalueren computer architectuur identiek zijn. Dit soort architectuur afhankelijke aannames zijn in onze ogen ongewenst. Ze ondermijnen het gemak waarmee simulatie modellen geconstrueerd worden of, nog erger, beperken de vrijheid van modelleren. Flexibiliteit in computer architectuur modellering is echter een noodzaak. Dit wordt onderstreept door de diversiteit van de drie case studies die in dit proefschrift worden beschreven.

In dit proefschrift introduceren we een nieuwe simulatie methodologie waarmee zowel goede simulatie snelheden als een hoge mate van flexibiliteit verkregen wordt. Deze simulatie methodologie, *operatie-gedreven simulatie* genaamd, is een combinatie van twee populaire simulatie methoden: stroom-gedreven simulatie en executie-gedreven simulatie. In operatie-gedreven simulatie maken we gebruik van een stroom van simulatie stimuli, *operaties* genaamd, die het applicatie gedrag weergeeft en de architectuur simulator aandrijft. De operaties kunnen zo worden gedefinieerd dat hun abstractie nivo goed aansluit op de architectuur onderdelen waarin de architect specifiek is geïnteresseerd. Ze kunnen bijvoorbeeld (abstracte) machine instructies representeren om de prestatie analyse met een hoge mate van nauwkeurigheid uit te voeren, of de operaties kunnen computatie en communicatie op een veel hoger abstractie nivo (bijvoorbeeld op het nivo van grofkorrelige taken) weergeven om een minder nauwkeurige maar snellere simulatie te bewerkstelligen. Met behulp van de laatste optie zouden we snel een ruw inzicht kunnen krijgen in de prestaties van een nieuwe computer architectuur. In operatie-gedreven simulatie wordt de toepassing van verschillende abstractie nivo's dus als instrument gebruikt om de snelheid (respons-tijd) en nauwkeurigheid van de simulatie te controleren.

Operatie-gedreven simulatie biedt een hoge mate van modelleer flexibiliteit. Hiervoor zijn twee redenen. Allereerst kunnen de operaties zo gekozen worden dat ze onafhankelijk zijn van zowel de architectuur van de gastheer computer als van de te modelleren architectuur. Dit houdt in dat er bijvoorbeeld meerdere processor-typen (met verschillende instructie sets) gesimuleerd kunnen worden zonder de architectuur simulator aan te passen. Ten tweede is het mogelijk om het modelleren van applicatie gedrag (het genereren van de operaties) los te koppelen van de architectuur modellering. Hierdoor kunnen de operaties op verschillende abstractie nivo's en met verschillende maten van nauwkeurigheid gegenereerd worden. We kunnen bijvoorbeeld het gedrag van een echte applicatie volgen om zodoende een realistische stroom van operaties te genereren. Of we kunnen de operaties simpelweg door een stochastisch proces laten genereren. Dit laatste is flexibel (het applicatie gedrag kan makkelijk worden aangepast) maar is tevens onnauwkeurig (het representeert vaak geen realistisch applicatie gedrag).

In hoofdstuk 3 wordt de Mermaid simulatie omgeving besproken waarin we de operatie-gedreven simulatie methode hebben toegepast. Deze omgeving biedt een simulatie raamwerk voor het uitvoeren van prestatie analyses van multicomputer architecturen. Hoofdstuk 4 toont aan dat de flexibiliteit van Mermaid, verkregen door de operatie-gedreven simulatie, de snelheid en nauwkeurigheid van de simulatie niet in gevaar brengt. Validatie experimenten hebben aangetoond dat er, ondanks het tamelijk hoge abstractie nivo van de simulatie modellen, nauwkeurige prestatie voorspellingen worden gedaan. Ook hebben we laten zien dat Mermaid op het gebied van simulatie snelheid goed kan concurreren met andere moderne simulatoren. Om Mermaid's simulatie snelheid nog verder te verbeteren, hebben we functionaliteit toegevoegd om de simulatie in meerdere deeltaken op te delen en deze taken over een cluster van werkstations te distribueren zodat ze in parallel kunnen worden uitgevoerd. We hebben door middel van experimenten laten zien dat deze gedistribueerde simulator sneller is dan bijna alle andere beschikbare parallelle architectuur simulatoren.

Hoofdstukken 5 tot en met 7 beschrijven drie case studies waarin we Mermaid hebben toegepast. In de eerste studie wordt een zogenaamd "wormhole-routed Mesh of Clos" netwerk geëvalueerd met behulp van een synthetische communicatie last. De tweede studie zoemt in op het prestatie gedrag van één enkele processor. Hierin onderzoeken we een aantal methoden om gegevens zo vroeg mogelijk (voordat ze werkelijk nodig zijn) in de data cache van een VLIW processor te laden. Dit wordt "prefetching" genoemd. In de laatste case studie wordt het prestatie gedrag van een schaalbare, gedistribueerde data structuur voor een multicomputer architectuur onderzocht.

De diversiteit van de case studies tezamen met het feit dat we Mermaid eenvoudig hebben kunnen toepassen in alle drie de studies illustreert de flexibiliteit van onze simulatie methodologie. Het laat tevens zien dat het met Mermaid mogelijk is om een prestatie analyse uit te voeren op zowel het architectuur nivo (hoofdstukken 5 en 6) als op het applicatie nivo (hoofdstuk 7).

In het geval van twee case studies waren we in staat om onze simulatie resultaten te valideren met de resultaten afkomstig van een echte machine of van een zeer nauwkeurige simulator. Deze validatie experimenten bevestigen de belangrijke conclusie uit hoofdstuk 4, namelijk dat een redelijk abstract model toch een goede simulatie nauwkeurigheid kan opleveren.

Index