In Proceedings of the 26th Annual Simulation Symposium, March 29-31, 1993
Arlington, U.S.A., pp. 41 - 49.

The CacheMire Test Bench — A Flexible and Effective Approach for Simulation of Multiprocessors

Mats Brorsson, Fredrik Dahlgren, Håkan Nilsson, and Per Stenström
Department of Computer Engineering, Lund University
P.O. Box 118, S-221 00 LUND, Sweden

Abstract

The approach of program-driven simulation of multiprocessors has generally been believed to be too slow in order to perform experiments and performance evaluations with realistic workloads.

We show that the program-driven approach for building multiprocessor simulators is indeed a viable method. It compares well in performance to an execution-driven simulator which has been reported in the literature, and has superior flexibility.

The reported simulator is the core in the CacheMire test bench which is an entire environment for conducting performance evaluations on shared memory multiprocessors. The test bench is used in a number of projects, including cache coherence protocol evaluation, super-pipelined processor design and analysis of parallel program behaviour.

1 Introduction

During the development of a new computer system, performance evaluations have to be made continuously. In the early stage of development, *analytical models* are often used to get a coarse estimation of performance. However, because of the difficulty in capturing the dynamic behaviour of the system, the applicability of this technique is so far limited to either very simple systems or for rough estimates.

To build *prototypes*, on which measurements are done, gives the most data on real performance. However, the design considerations must already have been made when the building starts, so this approach is mostly used at the final phase of the development.

Therefore, when investigating various different architectural features, *simulation modelling* is often used for performance evaluation. A simulation model can be made accurate, but still flexible enough to investigate a large design space.

Multiprocessor architectures, which have a potential of providing cost-effective performance, are a lot more difficult to model than single processor systems due to the very complex interaction between the architecture and the parallel program behaviour. Even though the approach of simulation has been widely used for multiprocessor performance evaluations, the simulation tools used for single processor systems have in most cases not provided enough information to perform accurate evaluations, or been effective enough to use realistically sized workloads.

In this paper we present the CacheMire test bench for performance evaluations and measurements of multiprocessor architectures. The core of the test bench is a flexible program-driven simulator, efficient enough to execute entire parallel programs as workload. The test bench has shown itself to be a very useful tool for quantitative measurements of shared memory multiprocessor architectures.

1.1 Background

We will in the following consider shared memory multiprocessors as target architectures. The concepts of simulation discussed in the paper are, however, independent of the memory model.

With modern processor architectures, the most crucial part of a shared memory multiprocessor is the memory system. For a high processor utilization, the memory system must be able to serve the processors with several tens of millions memory references per second and processor.

A simulation model for performance evaluation of memory systems for multiprocessors generally consists of: (i) a memory reference generator for each processor and (ii) a simulator of the memory system. Four techniques exist for generating the memory references [11]:

- *Distribution-driven simulation*. The workload is modelled by a stochastic model of the distribution of memory references.
- *Trace-driven simulation*. A trace of memory references is generated *once* by means of executing the workload on a machine similar to the target system or by using a functional simulator.
- *Execution-driven simulation*. The workload is executed on a host computer. At events of interest, e.g. shared memory references, control is transferred to the simulation software, which simulates the memory system.

- *Program-driven simulation*. Both processors and the memory system of the target system are simulated. The workload is executed on the simulated processors.

The method of distribution-driven simulation suffers mainly from the lack of good stochastic models of real parallel programs. In addition, this method, as well as the trace-driven method, does not have any feedback between the memory system architecture and the program behaviour. This may lead to considerable discrepancies in performance estimation compared to execution- and program-driven simulation since in these cases the workloads are allowed to behave differently as a consequence of changes in the timing of memory references [7].

Due to the simulation of entire instruction set processors, program-driven simulation is very computation intensive. Work on simulation techniques has therefore lately been focused on *execution-driven simulation* [9, 15, 22]. The processes of the workload application program are in this case executed (pseudo)-concurrently as separate threads of control on a host computer system. This execution is without slow-down (except for the lack of parallelism). At predefined events of interest, such as a shared memory reference, control is transferred to the simulation software which synchronises all processes of the workload and simulates the memory system.

This approach has the big advantage of being very efficient if the application program can execute for long periods between global events of interest. This advantage is highly dependent on the sharing behaviour of the application program. The main drawback is the lack of flexibility. The processor architecture is fixed to the one of the host computer and the operating system has to be modified in order to transfer control from the user application program to the memory system simulator. If the number of instructions between each global event is small, the efficiency benefits of execution-driven simulation will diminish.

In *program-driven simulation* (sometimes referred to as instruction-driven simulation), each instruction of the application program is interpreted by an instruction set simulator of the processor. While this approach has been used a few times in the past [2, 19], it has generally been considered too time consuming for execution of entire workloads due to the interpretation of every instruction. However, with the emergence of high performance and cost-effective computers, such as the common workstation, combined with the simpler instruction sets of modern microprocessors, this approach has started to gain interest. It is the most flexible and accurate approach of the four techniques. Changes in the processor architecture can easily be made and the timing model can be made arbitrarily precise.

1.2 Motivation

The main motivation behind the CacheMire test bench is based on the type of experiments that was anticipated to be carried out.

At the department of Computer Engineering at Lund University, one of the main research topics is focused on shared memory multiprocessors. Within this topic, research is conducted in diverse areas; from coherent cache systems, compiler design, and consistency models to parallel program behaviour. Experimental methods are used for performance evaluations and measurements. Because of this diverse nature of research areas, a flexible but still powerful tool was needed to satisfy various kinds of experiments.

Both execution- and program-driven simulation techniques were considered to fulfil the needs of accuracy. While it has been demonstrated clearly that execution-driven simulation can be done with enough efficiency to execute entire applications of reasonable sizes, the same had to be verified for program-driven simulation which generally has been considered to be too time consuming. Recent techniques for efficient instruction set architecture simulation have shown that it is feasible to simulate the processors as well as the memory system [4].

The flexibility issue that is most important is the ability to simulate just as much as needed for a particular study. Secondly, it is important to be able to experiment with various processor architectures. It should also be possible to configure the test bench for new experiments without too much difficulty. In order to utilize the available computing resource, consisting of a network of workstations from various manufacturers, the test bench has to be portable.

All this together, it was considered most appropriate to develop a program-driven simulator as the core of the test bench. Such a simulator is easily implemented as a single program which does not require additions in the operating system. In contrast, the operating system has to support transfer of control between the workload programs and the simulator when using execution-driven simulation. Program-driven simulation also opens up for realistic experiments with different instruction set architectures in multiprocessor environments, something which is impossible with execution-driven simulation.

The result, the CacheMire test bench, is an environment for conducting experiments with shared memory multiprocessor architectures. The minimal simulator implements a shared memory multiprocessor with a number of processors issuing memory references to a shared memory. The processors are instruction set simulators of the SPARC[1] processor. The user can either accurately model memory accesses for all memory references, including instruction

1. SPARC is a registered trademark of SPARC International, Inc.

fetches, or only model data references. By tagging areas of memory with user defined attributes, accesses to any specific memory area can be traced.

The rest of the paper is organised as follows: The next section describes the structure of the program-driven simulator and the application programming environment. Section 3 describes how this basic simulator model easily can be extended to accommodate accurate simulation of complex memory systems. Section 4 presents some performance measurements of the simulator. Some ongoing research projects using the CacheMire test bench are briefly reviewed in Section 5 and the paper is concluded after a summary.

2 Structure of the CacheMire test bench

The CacheMire test bench consists of a minimal program-driven simulator and a programming environment. The minimal simulator is divided into a processor kernel, a simulated memory and a multiprocessor framework.

We will in this section describe the various parts of the test bench including a short description on how the test bench is used for practical experiments.

2.1 Processor kernel

The processor kernel is a highly optimized instruction set simulator of the SPARC processor architecture [20]. The interface of a processor consists of the *processor id* of the current processor, the *virtual address*, *32-bits data word*, *memory operation* (Read, Write, or atomic Read-Modify-Write (RMW)), and a *byte mask* (specifying which bytes, within a word, will be written in a byte or half-word store operation).

The instruction set simulator is written in C and implements all instructions generated by the Sun[2]/SPARC-version of the GCC compiler. A few optional instructions such as the integer multiply/divide and square root instruction are not implemented. Some of these are used for implementing synchronisation primitives.

The processor kernel is operated by means of two C-functions: `InitSparc` and `SparcCycle`. `InitSparc` initiates the processor by creating the state information area which contains register contents and other state information. The program counter is initiated with the start address of the first instruction to execute and the designated stack pointer is initiated to a user defined start value. A parameter to `InitSparc` determines whether instruction fetches are visible or if only data references will be seen by the memory system.

A *processor cycle* consists of an instruction fetch, execution of the instruction and an optional data reference. Execution of the application program is advanced by calling the `SparcCycle` routine. If instruction fetches are simulated, each invocation of `SparcCycle` advances the simulator to the next memory reference, instruction fetches included. A processor cycle may then require more than one invocation of `SparcCycle`. Between each invocation, the processor state stores information on whether the processor waits for an instruction or for data.

If instruction fetches are invisible, the processor is advanced a number of processor cycles until the next data reference. A return parameter gives information on the number of advanced processor cycles.

Three memory operations are required by the processor: *Read*, *Write* and *atomic Read-Modify-Write* (RMW) operations. The latter operation writes a value to a memory location and expects the old contents of the memory location in return to the processor in a single indivisible operation. It is implemented by using the opcode of one of the optional instructions.

2.2 Memory model

The simulated memory implements a shared linear address space whose size is determined during initiation of the simulator. The three memory operations required by the processor are supported by the memory model. The RMW operation is implemented as a combined read and write operation in the memory. The shared address space is, by default, logically divided into 4 kilo-byte page frames and ranges from address 0 to the maximum specified address. The page size can be changed when recompiling the simulator.

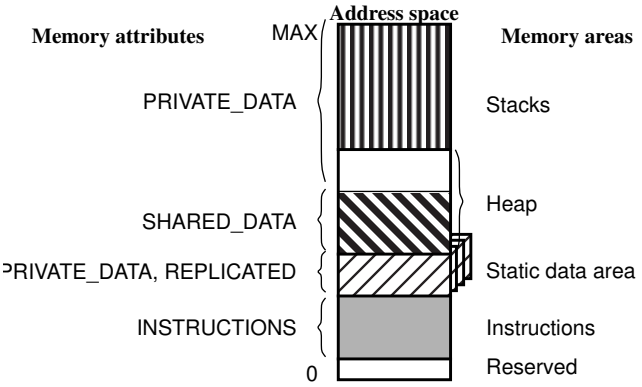

Figure 1. Memory map of the simulated memory.

Figure 1 shows the map of the memory model. The first two pages of memory are reserved for passing information from the simulator to the run-time environment of the application program. The instructions and the static data area are put in the bottom of the address space. The top portion is reserved for the processor stacks. Each processor gets a stack area the size of which is defined upon invocation of the simulator. The area inbetween is allocated to the dynamic memory area, the heap.

2. Sun is a registered trademark of Sun Microsystems, Inc.

The user (both the simulator user and the application programmer or compiler) is provided with primitives for tagging pages with attributes. These attributes are primarily used to distinguish pages in one of three sections of the address space: *instructions*, *private data* and *shared data*. A particular page may not belong to more than one of these sections.

A fourth attribute, *replicated*, specifies that a processor accessing a page with this attribute will get its own physical copy of the page. When a processor first touches a page with the attribute 'replicated', a page frame is allocated and the contents of the page is copied to the new physical page. Subsequent accesses to this page will be redirected to the unique physical copy of this page. No coherence is maintained on replicated pages so write operations to them will only be seen by the same processor that made the write.

The only exception from the linear address space is the static data area which the programming model requires to be replicated to all processing elements (further explanation in Section 2.4). Accesses by different processors to the same virtual address in the static data area will result in accesses to physically separate memory locations. This is achieved by tagging the static data area with the attribute 'replicated'.

2.3 Multiprocessor framework

The simulation of the execution of a parallel application program starts with an initiation phase in which the binary image of the application program is read and loaded into the simulated memory. The processors are initiated with the start address as stated in the binary file. Only processor 0 is executing from start and the application program is responsible of invoking the other processors as needed.

The framework for simulating a shared memory multiprocessor is an endless loop in which one iteration represents one processor cycle, which is the basic measure of time in the simulator (see Figure 2).

The first operation in an iteration, is to advance all processors having work to do. A time-stamp for each processor determines the earliest time it can proceed. A processor can during one iteration issue one instruction fetch and an optional data reference, or only the next data reference depending on the level of detail simulated. The time-stamp for the processor is updated to the earliest time it may proceed. Together with this, some operations which can be satisfied within a processor cycle may be done. For example, if the latency times of a memory reference can be statically determined, it may be satisfied at this point.

After all the processors have had the opportunity to proceed, other functional units are scheduled if they have work to do during this time step. Examples of such functional units are memory buffers and memory modules serving the processors with memory references. Another example is an interconnection network.

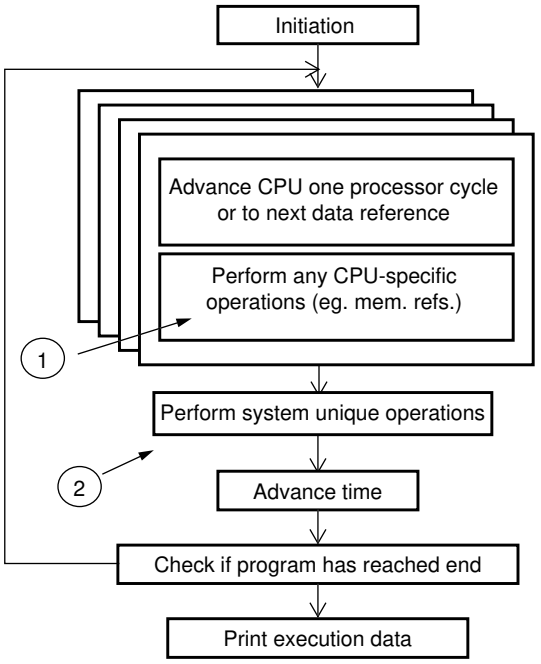

Figure 2. Multiprocessor framework of the minimal simulator

The global time is maintained and updated for each iteration of the loop.One time step represents the minimum time for one processor cycle.

The minimal simulator implements a shared memory multiprocessor with instantaneous access to memory. The memory references are performed at number 1 in Figure 2.

2.4 Programming environment

The programming environment consists of a run-time library, a generic trap function, a C-compiler and a macro package used for expressing parallelism.

A run-time library has been developed to support the execution of the application programs. Many of the standard run-time library functions such as I/O and the mathematical functions are implemented using traps as described above. Other functions, such as the memory allocation routines `malloc` and `calloc` are implemented directly in C.

The generic trap function is used for passing information from the application program to the simulator and vice versa. An example is the tagging of attributes to regions of memory mentioned earlier. The routines for tagging pages with attributes are implemented in the simulation software, but can be invoked from the application program by means of traps. The tag operation is provided with a trap code and the parameters to the tag function are put on the call frame as for any normal function call. When the trap instruction is executed, the call frame of this processor is examined by the simulator program and a call is made to the function implementing the association of attributes to memory.

The CacheMire test bench supports at the moment parallel programs written in C using the *Single-Program-Multiple-Data* (SPMD) model of computation [8]. In this model, all processors execute the same program but operate on different data which is scheduled during run-time. The ANL macros from Argonne National Laboratories are used for expressing the parallelism [5]. The machine dependent part of the ANL macro package has been changed to fit our run-time environment.

The definition of the ANL macros specifies that the static data area be replicated so that each processor gets its own physical copy. Some programs written using the ANL macros have been found relying on the replication of the static data area when the parallel threads of computation have been created. The creation of a parallel thread invokes a new processor which will inherit the values of the static data area as set up by the master processor, see Section 2.2. This behaviour is ensured by tagging the static data area with the 'replicated' attribute.

2.5 Using the CacheMire test bench

Inserting memory system simulators: Detailed simulators of memory systems can be inserted in the framework at positions corresponding to numbers 1 and 2 in Figure 2.

The memory references are inserted into the memory system at number 1. If the modelling of a memory reference cannot be done statically, the memory reference is inserted in a reference buffer and the processor has to be stalled until the memory reference is completed. This is achieved by keeping the stall-status in the processor state which prevents the processor from advancing until the stall-status is cleared.

At number 2 in Figure 2, the modelling of other functional units is done. The reference buffers of each processor are examined and the actions of the simulated memory system and network, due during this time step, are performed. For correct timing a timestamp is associated to each memory reference for determining the earliest time it may arrive to a functional unit in the simulated architecture.

Invoking the simulator: The simulator software and the application programs (the workload) are separate entities, in contrast to some other simulation techniques in which the simulator software is linked together with the application program. Once a simulator program has been compiled, any application program which has been compiled with the CacheMire run-time library may be used as workload.

Most of the workload programs we use come from the Stanford SPLASH-suite of parallel programs [18]. The SPLASH-suite is a set of both scientific and engineering applications which have been developed primarily for solving a scientific or engineering problems, not for performing computer performance evaluation.

There are a number of run-time options to the simulator program upon invocation. There are switches for specifying the size of the memory, the size of the stack each processor gets, the number of processors and whether instruction fetches are simulated or not. A simulation run is started by invoking the simulator with appropriate parameters. Tha last parameter should be the full name and parameters of the application program.

The next section describes how a detailed memory system and network simulator easily can be incorporated without sacrificing too much performance.

3 Example of a CC-NUMA simulator

In order to demonstrate the flexibility and effectiveness of the CacheMire test bench, we will now describe one example of how the test bench has been used.

As part of a masters thesis project, a detailed memory system and network simulator was designed and incorporated with the minimal simulator described in the previous section [13]. This simulator implements a shared-memory cache coherent non-uniform memory access (CC-NUMA) multiprocessor with a double two-dimensional mesh as interconnection network. The cache coherence protocol in the CC-NUMA simulator is a full-map write-invalidate protocol based on the Stanford DASH multiprocessor [14]. Figure 3 shows the structure of the simulated system.

The additions made to the minimal simulator were two modules implementing the coherent cache memories and the mesh networks. For data references in the shared region of the address space, a call to the cache memory was inserted at number 1 in Figure 2. The processor is free to advance to the next memory reference if there is a hit in the cache memory. Otherwise, the processor is stalled until the cache block containing the data is fetched to the cache memory according to the cache-coherence protocol. The simulator implements the weak ordering memory consistency model [10] which means that the processor does not have to wait for acknowledgement of write operations. In this consistency model the processor is allowed to issue write operations as long as a synchronisation point is not encountered.

When all processors have been scheduled once (some may issue a memory reference and some may be stalled due to read misses), the local memories and the network are scheduled. Correct latency times are modelled by tagging each buffer entry with a time-stamp and not scheduling messages in the network until they are due according to their time stamp.

The CC-NUMA simulator has been used for a number of research studies, including a comparison between write-invalidate and write-update cache coherence protocols [13, 21].

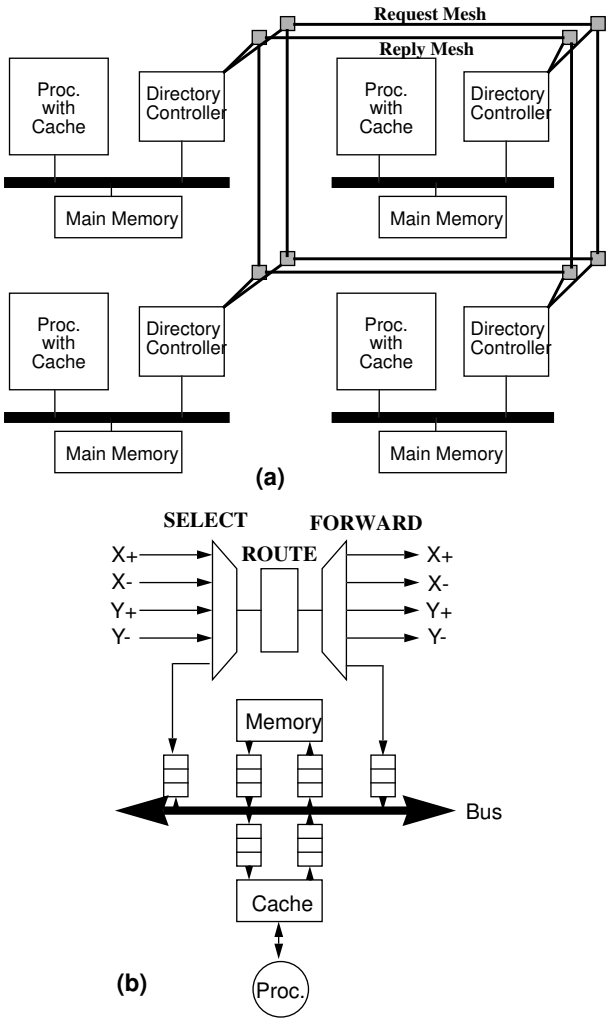

Figure 3. (a) Block diagram of a 2 x 2 CC-NUMA system. (b) Structure of a simulated processing node.

4 Performance of the simulator

This section presents some measurements regarding the performance of the program-driven simulator in the CacheMire test bench.

4.1 Performance measurements

For measuring the performance of the simulator, we have used two programs: MP3D from the Stanford SPLASH suite [18], and MATMUL which is a simple program multiplying two matrices of floating point numbers. MP3D is a program simulating a body in a flow of very low-density particles.

Table 1 shows some characteristics of the two applications. The numbers on the amount of instructions and data references are measured using the minimal simulator with 16 processing elements. It corresponds to the execution of the entire applications including a sequential initiation. For MATMUL the initiation phase is less than 10% but for MP3D it can be up to 50% of the total execution time. This initiation phase grows for MP3D with the number of particles but diminishes with the number of time steps.The distribution of memory references between instructions, private and shared data will of course vary depending on the configuration of the simulated system.

Table 1. Application characteristics

	MP3D	MATMUL
code size (lines)	1735	113
data set	10000 particles 10 time steps	128×128 elements
execution time (uniprocessor)	2.6 s	1.5 s
number of instructions	$27.6 \cdot 10^6$ (78 %)	$27.5 \cdot 10^6$ (76 %)
number of private data reads	$2.6 \cdot 10^6$ (7.5 %)	17240 (0.05 %)
number of private data writes	$1.16 \cdot 10^6$ (3.3 %)	125 (0.0 %)
number of shared data reads	$2.47 \cdot 10^6$ (7.1 %)	$6.3 \cdot 10^6$ (17.5 %)
number of shared data writes	$1.5 \cdot 10^6$ (4.3 %)	$2.1 \cdot 10^6$ (5.8 %)
number of atomic RMW	16554 (0.05%)	684 (0.0%)

We have evaluated the performance of the minimal simulator, which will give us the overhead of the processor simulator together with the multiprocessor framework. We have also, as a comparison, evaluated the performance of the CC-NUMA simulator described in Section 3. In order to conduct the performance evaluation we have made two sets of experiments. One set in which we vary the number of simulated processing elements from 1 to 64, and one set where we use the profiling utility of the GCC compiler in order to find which parts of the simulator contributes the most to simulation time.

All measurements were conducted on a Sun SPARCstation ELC peaking at approximately 25 MIPS. Both the simulator and the application programs were compiled using the GCC compiler and the -O2 optimization flag.

4.2 Results

Figure 4 shows the slowdown of the simulated multiprocessors, compared to execution of the application program on a Sun SPARCstation ELC single processor system. The slowdown for the minimal simulator ranges from 100 to 300 depending on the number of simulated processors and application program. This overhead is very small and it does not vary appreciably with the number of simulated processing elements. The average slowdown is around 200 which means that on average not more than 200 host instructions are needed to simulate an instruction fetch

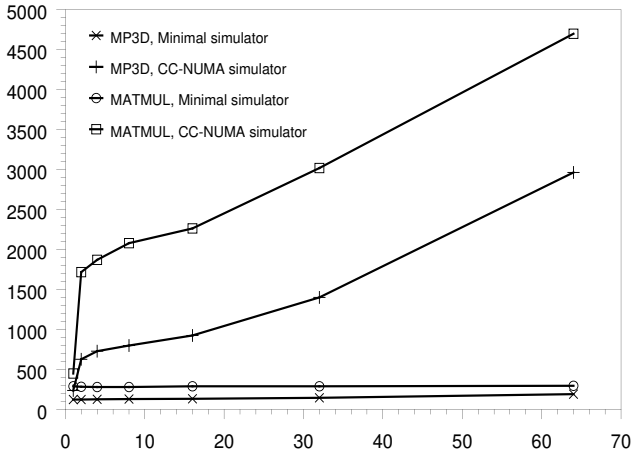

Figure 4. Slowdown of simulation of varying number of processing elements compared to an execution on a SPARCstation ELC single processor.

(with address translation), the interpretation of the instruction, possible data reference (also with address translation), reference counting and the framework for multiprocessing. The total time for simulating one complete instruction is about 16 µs (see Table 2).

The slowdown for the CC-NUMA simulator is included for comparison. The overhead of simulating caches, the mesh networks and the cache coherence protocol results in a lot higher slowdown as we increase the number of simulated processors. This memory system simulator suffers much from the same effects as in a real system. When we add processing elements the network will at some point become saturated and most of the time will be spent in routing messages through the network. Therefore, we get a more rapid increase in simulation time when the number of simulated processors exceed 16.

Table 2. Time for performing central operations in the simulators. 16 simulated processing elements.

	Minimal simulator	CC-NUMA simulator
Average time for one instruction	7 µs	8 µs
Address translation	2 µs	2 µs
Memory reference	3 µs	3 µs
Main loop overhead	4 µs	11 µs
Sum	16 µs	24 µs

Table 2 shows the times for performing central operations and how they vary between the minimal and the CC-NUMA simulator. On top of this comes the time for simulating the memory system. The differences between the both simulators are in the main loop and in the processor simulator. The difference in the main loop is because of the additional work which has to be done when more functional units are added to the simulator. The difference in the average time for simulating the execution of one instruction can be attributed to the changes in dynamically executed instructions between the two experiments.

The low overhead of simulating the processors compares well in performance with other simulation systems, such as the Tango system from Stanford [9]. Tango is an execution-driven simulator (the application program is executed on the host computer, *not* by a functional simulator) to which a number of memory system simulators can be attached. Depending on the kind of memory simulator being used, the amount of overhead varies, but the simplest memory model called *"Single-Issue"* causes at least 20 µs overhead for each global event being traced [9]. Other memory simulators lead to a higher overhead which is in the order of 300-700 µs because of expensive context switches. The global events which can be traced by Tango are synchronisation operations, accesses to shared data and accesses to private data. Instruction fetches are thus invisible for Tango.

Assume that we are only interested in shared data and that 17% of the instructions lead to a shared data access (average for MP3D and MATMUL). Assume further that we are using a host computer which is able to execute instructions at an average rate of 20 MIPS. The average minimum time for executing one instruction with Tango is $0.17 \cdot 20 + 0.05 = 3.45$ µs. With more sophisticated memory simulators in Tango, this time is: $0.17 \cdot 300 + 0.05 = 51$ µs.

The corresponding time for the CacheMire test bench is 16 µs, and in this case *all* memory references are observed. We can thus conclude that while the CacheMire test bench is not as fast as the fastest memory simulator on Tango, the basic time needed for simulating processor and memory references is very well acceptable and less than most of the published overhead numbers for Tango.

Figure 5 breaks down the execution to the most time consuming operations in the simulators. The figure includes measurements done on the minimal and the CC-NUMA simulator executing MP3D with 16 processors.

From Figure 5 we see that even for the minimal simulator, less than 40% of the simulation time is spent interpreting instructions. For the CC-NUMA simulator it is less

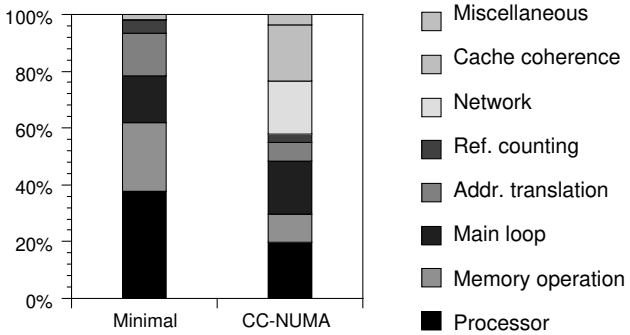

Figure 5. Sources of execution time in the minimal simulator and CC-NUMA simulator with 16 processing elements.

than 20%! The main contribution to simulation time is the memory system, a conclusion also observed by Davis at al. in [9].

In summary we have seen that it is very well feasible to use the approach of program-driven simulation for evaluation of multiprocessor architectures. The main sources of execution time when performing detailed simulation of memory systems are besides the processor simulation, the network and the multiprocessor framework (the main loop).

5 The use of CacheMire test bench in research

This section describes a few projects in which the CacheMire test bench is used as a major tool for the investigations.

5.1 Visualisation of data sharing

Many researchers have identified the sharing behaviour of shared-memory parallel programs as one of the key issues for fast execution. Given a shared memory multiprocessor architecture, depending on the sharing behaviour of the workload, the performance can vary drastically.

This study [6] uses the CacheMire test bench in a methodology of capturing and visualising the sharing behaviour of parallel applications. The stream of shared memory references generated by the processors are analysed during run-time and key parameters describing the sharing behaviour of the program are recorded.

5.2 Evaluation of new cache-coherence protocols

The CacheMire test bench is especially suitable to aid in evaluation of new cache-coherence protocols and architectures.

A number of link-based cache coherence protocols are evaluated in [16]. One of them is a new tree-based protocol presented in [17]. The minimal simulator in the CacheMire test bench has been augmented with a generic contention free network, which simulates accurate latency times, and with processor caches which are kept coherent using one of the evaluated cache-coherence protocols.

The relative performance differences of different network bandwidths and latencies are evaluated for both a write-invalidate and a write-update cache-coherence protocol in [13]. This study incorporated the augmentation of the minimal simulator with the cache-coherence protocol of the Stanford DASH multiprocessor (see Section 3). In contrast to the study mentioned above, this memory system simulator accurately simulates the effects of contention in the network and memory modules.

5.3 Experiments with new processor architectures

A master's thesis project at the department has dealt with the design and evaluation of a non-blocking read algorithm for a SPARC processor [12]. Normally, a processor always stalls on a read-miss in the cache memory. This project has used the CacheMire test bench to evaluate a new proposal on how to avoid this problem.

5.4 Trace generation

The minimal simulator has been used in a couple of projects for generating traces.

Cache-coherence protocols on slotted rings: A project at the department of Electrical Engineering Systems at University of Southern California is performing an evaluation of cache coherence protocols on a slotted ring. The evaluation is based on detailed trace driven simulations of both the cache coherence protocols and the ring interconnection. A variety of cache coherence protocols and ring architectures are investigated. The CacheMire test bench is used to generate the traces in this study [3].

Instruction mix measurements: The single-processor version of the CacheMire test bench has been used to measure the relative occurrences of instructions in ordinary sequential programs such as TeX, Spice and GCC. The information is used in the design and construction of a highly-pipelined processor with a target clock frequency of 500 MHz or more [1].

6 Summary and conclusions

Program-driven simulators are much more flexible than their execution-driven counterparts. They allow experiments with different processor architectures in multiprocessor environments, and let the user control all parameters such as processor cycle times and the number of outstanding references to the memory system. They can also easily be built portable to be used in multi-platform computer networks.

We have shown that with modern computing resources, detailed program-driven simulation of multiprocessor architectures is indeed a viable approach for performance evaluations, in contrast to what has generally been believed.

We have in this paper described the CacheMire test bench which is based on a program-driven simulator of a shared memory multiprocessor. The performance of this simulator compares well with the performance of Tango, an execution-driven simulator developed and used at Stanford University. When performing detailed simulation of memory systems, the time needed for interpretation of instructions, becomes a smaller issue compared to the need for efficient simulation of the memory system itself.

The CacheMire test bench is particularly well suited to be used in a computing environment with a network of workstations because of its simplicity and portability. A particular study often requires a series of experiments, and in a networked environment each experiment can be submitted to different workstations thus achieving a high degree of parallelism in the experiments.

The CacheMire test bench has been used in a number of studies and has proved itself to be a powerful tool for performance evaluations. The test bench supplies the user with a minimal simulator with no simulation of a memory system. This has to be added by the user and can in its simplest form consist of analytical computation of latency times. Experiences from several research projects have shown that it is easy to add a memory system to the minimal simulator.

In spite of the effectiveness of the CacheMire test bench its main drawback is just the need for computational power. Many interesting experiments are too time-consuming in order to be practical to perform, e.g. experiments on operating systems. The solution to this is not other simulation techniques such as execution-driven simulation. Rather will the development of new high-performance and cost-effective computing resources lead to the feasibility of program-driven simulation also in this area.

Acknowledgements

Magnus Karlsson is gratefully acknowledged for implementing the CC-NUMA simulator.

This work was supported by the Swedish National Board for Industrial and Technical Development (Nutek) under contract number 9001797.

References

[1] P. Andersson et al., *A Superpipelined Strategy for High Speed CMOS Microprocessors*. Tech. Rep., Dept of Comp. Eng., Lund University, Box 118, S-221 00 Lund, Sweden, Nov. 1992.

[2] T. S. Axelrod, P. Dubois, and P. A. Elgroth. A Simulator for MIMD Performance Prediction. *Parallel Computing*. pp 1237-274, 1984.

[3] L. A. Barroso and M. Dubois, *The Performance of Cache-Coherent Ring-based Multiprocessors*. Tech. Report, Dept. of EE-Systems, Univ. of S. Calif., USA, Nov., 1992

[4] R. Bedichek. Some Efficient Architecture Simulation Techniques. *Proc. of USENIX 1990 Winter Conf.*, pp 53-63. 1990.

[5] J. Boyle et al. Portable Programs for Parallel Processors. Holt, Rinehart, and Winston Inc. 1987.

[6] M. Brorsson, and P. Stenström. Visualising Sharing Behaviour in relation to Shared Memory Management. *Proc. Int. Conf. on Parallel and Distributed Systems*, pp 528-536, Hsinchu, Taiwan, Dec. 1992.

[7] F. Dahlgren. A Program-driven Simulation Model of an MIMD Multiprocessor. *Proc. of the 24th Ann. Simulation Symp.*, pp 40-49. 1991.

[8] F. Darema et al. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*. 7(1):11-24, April, 1988.

[9] H. Davis, S. R. Goldschmidt, and J. Hennessy. Multiprocessor Simulation and Tracing using Tango. *Proc. of the 1991 Conf. on Parallel Proc.*, pp 99-107. 1991.

[10] M. Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessor. In *Proc. of the 13th Ann. Int. Symp. on Comp. Arch.*, pp 434-442, 1986.

[11] D. Ferrari. *Computer Systems Performance Evaluation.* Prentice Hall, 1978.

[12] D. Gullberg. *A Non-Blocking Read Algorithm for a SPARC processor — Implementation and Evaluation*. Tech. Rep., Dept. of Comp. Eng., Lund Univ., Box 118, S-221 00 Lund, Sweden, Jan. 1992. MSc thesis

[13] M. Karlsson. *Bandwidth and Latency Implications of Directory-Based Cache Coherence Policies.* Tech. Rep., Dept of Comp. Eng., Lund University, Box 118, S-221 00 Lund, Sweden, Aug. 1992. MSc thesis

[14] D. E. Lenoski et al., The directory-based cache coherence protocol for the DASH multiprocessor. In *Proc. of the 17th Ann. Int. Symp. on Comp. Arch.*, pp 148-159, May 1990.

[15] R. Mukherjee., and J. Bennet. Simulation of Parallel Computer Systems on a Shared Memory Multiprocessor. *Proc. of the 23rd Hawaii Int. Conf. on System Sciences*. Jan. 1990.

[16] H. Nilsson, and P. Stenström. Performance Evaluation of Link-Based Cache Coherence Schemes. *Proc. of the 26th Hawaii Int. Conf. on System Sciences*. 1993.

[17] H. Nilsson, and P. Stenström. Scalable Tree Protocol - A Cache Coherence Approach for Large-Scale Multiprocessors. *4th IEEE Symp. on Parallel and Distributed Processing*. Dec. 1992.

[18] J. P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Computer Architecture News, 20(1):5-44, March 1992.

[19] K. So, F. Darema-Rogers, D. A. George, V. A. Norton, and G. F. Pfister. *PSIMUL - A System for Parallel Simulation of the Execution of Parallel Programs.* Technical Report RC 11674, IBM T. J. Watson Reseach Center, Yorktown Heights, 1986.

[20] SPARC International. *The SPARC Architecture Manual, Version 8.* SPARC International Inc., 535 Middlefield Rd, Suite 210, Menlo Park, Ca 94025., 1991.

[21] P. Stenström, M. Brorsson and L. Sandberg, *An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing*. Tech. Rep., Dept. of Comp. Eng., Lund University, Box 118, S-221 00 Lund, Sweden, Nov. 1992. Submitted for publication.

[22] C. B. Stunkel, and W. K. Fuchs. TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation,. *Performance Evaluation Review.* 17(1):70-78, May, 1989. Proc. of 1989 ACM SIGMETRICS and Performance '89.