

A New Program-Driven Parallel Machine Simulation Environment

PO-ZUNG CHEN AND SHYH-NONG CHEN*

*Department of Computer Science and Information Engineering
Tamkang University*

Taipei, Taiwan 251, R.O.C.

E-mail: pozung@cs.tku.edu.tw

**E-mail: csnong@cs.tku.edu.tw*

In recent years, it has gradually become popular to use discrete-event simulation as a tool for analyzing the hardware architecture of shared-memory multiprocessors. A complete and detailed machine simulation environment can be utilized to evaluate the performance of a completed prototype hardware architecture in an operating system and application software. This paper describes the development, operation, and future development of a parallel machine simulation environment which has the ability to execute programs. At present, a shared-memory multiprocessor simulator has been developed that can execute 80x86 programs. On the other hand, we have focused on parallel discrete-event simulation to reduce the simulation time. The simulator works through Ethernet to connect multiple workstations to solve problems. Some workstations are used for calculations while the others are used for memory management. Thus, it is easy to get a parallel machine simulation environment. This simulator can provide an environment which can be used to execute application programs, and to simulate different system architectures by adjusting parameters. In addition it can calculate the run time for an application program so as to evaluate the possibility of executing the application program in a real system. The source code for the simulator can be downloaded from <ftp://loon.cs.tku.edu.tw/pub/86sim.tgz>.

Keywords: shared-memory multiprocessors, multicache coherence, coherence protocol, discrete-event simulation, trace-driven simulation, instruction-set emulation

1. INTRODUCTION

Tremendous progress has been made in the development of computer hardware, so shared-memory multiprocessors have become very popular. Theoretically, the system which combines the most processors can provide the highest memory bandwidth and processor capability as well as the best raw parallelism. With shared-memory multiprocessors, due to the rapidly increasing number of processors, one goal is to utilize the connection architecture of the bus, crossbar, or multistage interconnection to efficiently connect processors, caches and shared-memory, and to maintain multicache coherence among these units. A further challenge is the scalability problem [1, 2].

Received April 2, 1999; revised June 30, 1999; accepted November 26, 1999.
Communicated by Shang-Rong Tsai.

Furthermore, as the gap between processor and memory speed continues to widen, methods for evaluating memory system designs before they are implemented in hardware are becoming increasingly important. The adoption of cache memories [3, 4] in multiprocessor systems results in two advantages: reduction of the average time needed to access shared-memory and minimization of the bus requirements for each processor. Both of these advantages yield a significant improvement in the global performance of the system.

In one commonly adopted scheme, each processor is supplied with a private cache, containing the data and code that are most frequently referenced by the processor. The caches provide a distributed and private image of the single, shared-memory. However, such a solution leads to the coherence problem [5-8]: When two or more processors store a copy of the same memory block in their private caches and one of them performs a write operation of a location within that block, a coherence protocol is required in order to guarantee that every subsequent read operation done by any processor may get the up-to-date value of the modified location. Detailed descriptions of the main existing protocols can be found in [5, 9, 10].

Many analytical models have been used to evaluate the performance of the bus-protocol in maintaining multicache coherence in shared-memory multiprocessors [11, 12]. However, due to the increasing complexity of computer systems, it is necessary to apply simplified assumptions to the analyzed system, which results in some preliminary limitations placed on the analytical results.

In recent years, it has gradually become popular to use discrete-event simulation as an tool for analyzing the hardware architecture of shared-memory multiprocessors. A complete and detailed shared-memory multiprocessor simulator can not only simulate the whole system, but also load real software on it. For example, the operating system [13] and a distributed database application program can be run on it just like on a real hardware machine. Thus, the performance of the real operating system and application software can be evaluated. Moreover, before the hardware is made, this facilitates the development, production, and improvement of the operating system and application software [14].

This paper describes the development, operation, and future development of a parallel machine simulation environment which has the ability to execute programs. At present, shared-memory multiprocessors simulator have been developed that can execute 80x86 programs. On the other hand, reduction of the simulation time by through parallel discrete-event simulation is also addressed here [15, 16]. The execution platform for the simulator is first set up by using multiple workstations connected by Ethernet.

1.1 Contributions and Related Work

The major contribution of the paper is to propose a parallel machine simulation environment that has the ability to execute programs. In the actual simulator, the technology for simulating a virtual computer system is provided so that the efficiency of the system can be analyzed and the results used as a reference for further improvement or for deciding on adoption of the system. This is meant to improve production efficiency and reduce R&D expenses. Moreover, we provide a parallel processing environment for executing application programs.

As simulated systems are becoming increasingly complicated, we provide two different technologies so that evaluation results can be derived in the shortest time. One is called machine emulation, which is fast but results in a loss of fidelity. By means of this method the simulator emulates the results after the computer system is operated only by means of

instruction translation, not by completely emulating the hardware operation, so information for evaluation may not be sufficient [17]. The other is machine simulation used to simulate the hardware operation precisely and to acquire detailed results [14].

Fig. 1 shows five classes based on implementing trace-driven simulation [18] (refer to Appendix A) at different system levels. This paper presents the simulation environment developed on the basis of instruction-set emulation. Instruction-set emulation uses the host (simulator execution platform) instruction set to execute the target (where the simulated hardware platform can be either an existing one or an unimplemented one) instruction set, so the simulator can be used to develop an unimplemented machine.

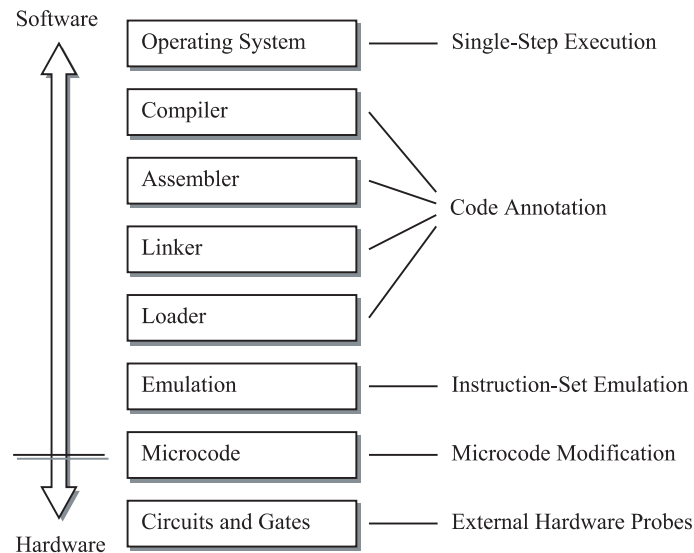


Fig. 1. Levels of system abstraction and trace-driven simulation.

The advantages of instruction-set emulation are ease to use, portability, and flexibility while the disadvantages are that it is slow, and that it is necessary to implement a large number of states, which make the design task more difficult (refer to Appendix A.1). When the simulator executes a target instruction, it uses more memory than does native code for translation into equivalent host instructions. Although instruction-set emulation will slow down and used extra memory, we think the high speed of today's microprocessors and the decreasing lost memory modules will make instruction-set emulation feasible. Furthermore, it does not need expensive equipment, and it is not limited to existing machines. These were our major considerations when we adopted instruction-set emulation to develop our simulation environment.

In the following, we list some well-know simulators that resemble ours in order to compare the advantages of our simulator with those of the others and to present the new design technology. We leave the detailed performance evaluation to Section 4. Below are the simulators that were designed on the basis of instruction-set emulation. As to other

methods (i.e. implementation methods for other levels in trace-driven simulation), due to the large differences between them, we will not offer any descriptions or comparison. Detailed information can be found in [18].

Spa [19] was proposed by Cmelik and Keppel in 1993. It was implemented on SPARC. It simulates SPARC, and the reported slowdown ranges from 40 to about 600.

Mable [20] was proposed by Davies in 1994. It was implemented on MIPS-I. It simulates MIPS-I and MIPS-III. The reported slowdown ranges from 20 to about 200.

SPIM [21] was proposed by Larus in 1991. It was implemented on systems like SPARC, 680x0, MIPS, x86, HP-PA, etc. It simulates MIPS-I, and the reported slowdown is about 25.

MINT [22] was proposed by Veenstra and Fowler in 1994. It was implemented on R3000. It simulates R3000 and is a shared-memory multiprocessor simulator. Veenstra reports slowdown for MINT in the range 20 to 70 from emulation of a single processor.

Shade [23] was proposed by Cmelik and Keppel in 1994. It was implemented on SPARC-V8. It simulates SPARC-V9, SPARC-V9, and MIPS. The reported slowdown ranges from 9 to 14.

All of these simulators collect memory references from only a single process, and these reported slowdowns should be interpreted carefully because a large slowdown may represent the time required to emulate processor activity that is not strictly required to generate memory references.

Our research is not very much related to the above topics (only similar on the basis of instruction-set emulation), so it is not easy to find a right way to evaluate. The major reasons are that they are mostly implemented on SPARC and the simulated objects are systems like SPARC and MIPS, etc. Our simulator can be operated on hardware like Sun SPARC, IBM RS 6000, Intel 80x86, etc. The simulated objects are shared-memory multiprocessors. We summarize the main characteristics of our simulator as follows: (1) It can be easily re-implemented on a UNIX-based system. (2) It uses two key technologies, instruction-set emulation and parallel discrete-event simulation, for the core part of the simulators. (3) It provides a parallel machine simulation environment. (4) It uses workstations in a network to solve a problem through cooperation, and it provides a parallel processing environment.

1.2 Overview of the Paper

The rest of this paper is structured as follows: Section 2 gives a description of the system and input/output parameters. Section 3 describes the overall implementation of the simulation environment. Section 4 presents the simulation and results. In Section 5 we draw conclusions and discuss further research. Appendix A gives a brief description of trace-driven simulation, and Appendix B presents an instruction-set emulation program segment. Finally, we add references and illustrations.

2. DESCRIPTION OF THE SYSTEM AND INPUT/OUTPUT PARAMETERS

The hardware system in question consists of N independent processors (CPU_i); each processor interacts with a private cache ($Cache_i$), and the caches may access a shared main memory (Shared-Memory) via a common bus (Bus). The purpose of our machine simulation environment is to do performance evaluation on different multiprocessor architectures and

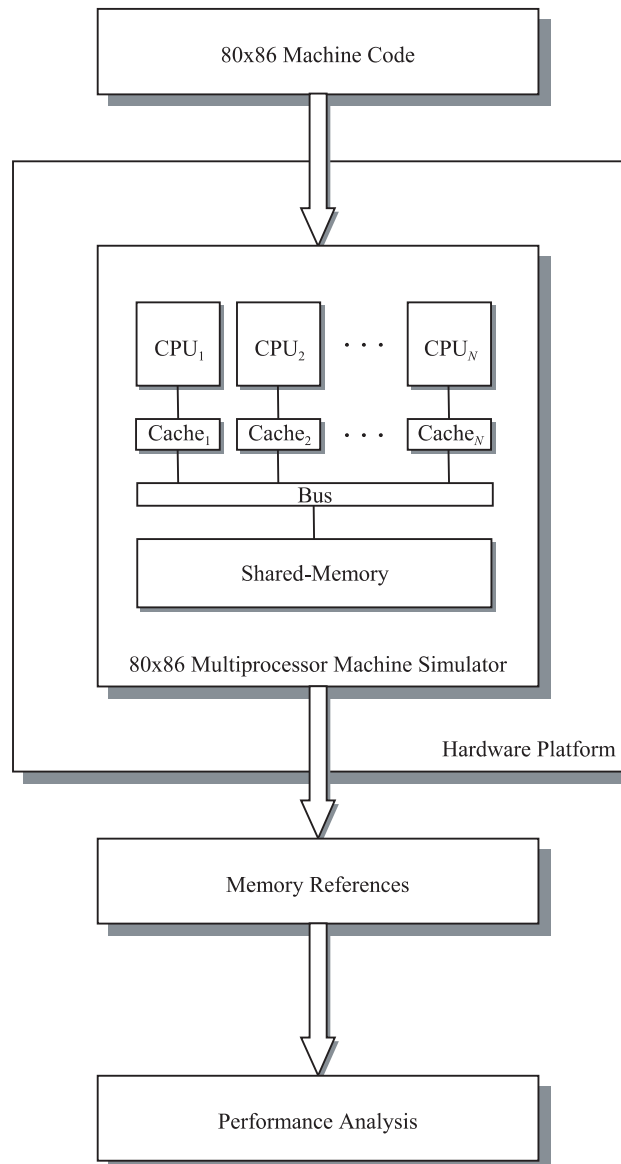


Fig. 2. 80x86 multiprocessor machine simulation environment.

execute application software under real operation conditions for operating systems and application software. As shown in Fig. 2, the whole machine simulation environment contains three levels. The 80x86 machine code, which produced by real workload, is at the top level (including complicated software like operating systems). The middle level is the core of simulator, which is composed of multiple 80x86 processors. It does simulation and also recording on memory references through the simulator, which is designed based on instruction-set emulation. It determines the parameters for a higher cache hit ratio to improve the

efficiency of the next identical execution on the same program by analyzing the memory references of the application software. The bottom level is the hardware platform for the whole simulator. It uses parallel discrete-event simulation to reduce the simulation time, so workstations connected by Ethernet are the execution platform for real operation at this time. It will be necessary to consider the use of multicomputers and multiprocessors as the execution platform. The simulator can also provide program debugging (working as a software debugger), simulate time calculation, and modify the simulated system architecture using provided parameters to further study the efficiency of the hardware architecture.

In the following section 2.1, we will introduce a way to design the simulator by using instruction-set emulation. In section 2.2, we will describe the implementation of parallel discrete-event simulation. In section 2.3, we will discuss synchronization under a distributed environment. In section 2.4, we will describe the input/output parameters provided by the simulator.

2.1 Instruction-Set Emulation

The core of the simulator is designed based on instruction-set emulation using the method of trace-driven simulation. It simulates the instruction set of the simulated system platform by executing the instruction set of the simulator hardware platform. The advantages are that it is easy to transfer the simulator designed using the C language to other hardware platforms, and that it is able to simulate an unimplemented system. (Now although the simulating system already exists, we can still add extra functions to do efficiency analysis at the beginning of the design process, which is also our purpose in designing the simulator). The core program is based on an endless loop, and by using C language switch statements, it can do instruction translation (refer to Appendix B) according to every operation code in an 80x86 processor. It keeps a record of the memory references throughout the input/output process.

The bottleneck of trace-driven simulation is in collecting and handling each memory reference from workload. Many trace-driven simulation tools can only make partial memory references without considering if the simulated memory states have changed. Thus, the result can be incomplete and inaccurate memory references with large amounts of unnecessary memory references, which consume much disk space and lead to inefficient analysis. Our simulator can find memory references, which do not affect the simulated memory states, so as to reduce the cost of analyzing the memory references, such as cache hits (which do not need to update the cache like cache miss).

2.2 Parallel Discrete-Event Simulation

It uses an existing computer system to simulate the same computer system with different characteristics through instruction-set emulation, and even to simulate an unimplemented computer system. However, we want to know how to reach synchronization among processors when a multiprocessor system is to be simulated. To simulate multiprocessors, we add discrete-event simulation to our simulator design to generate every kind of event through instruction translation. We assume that the system is composed of a collection of processes. Each process consists of a sequence of events.

80x86 multiprocessor simulator is implemented according to discrete-event simulation as shown in Fig. 3. Basically, the simulator takes the first event to simulate and calculates the simulating time from pending-event set. It adds a new event to the pending-event set if

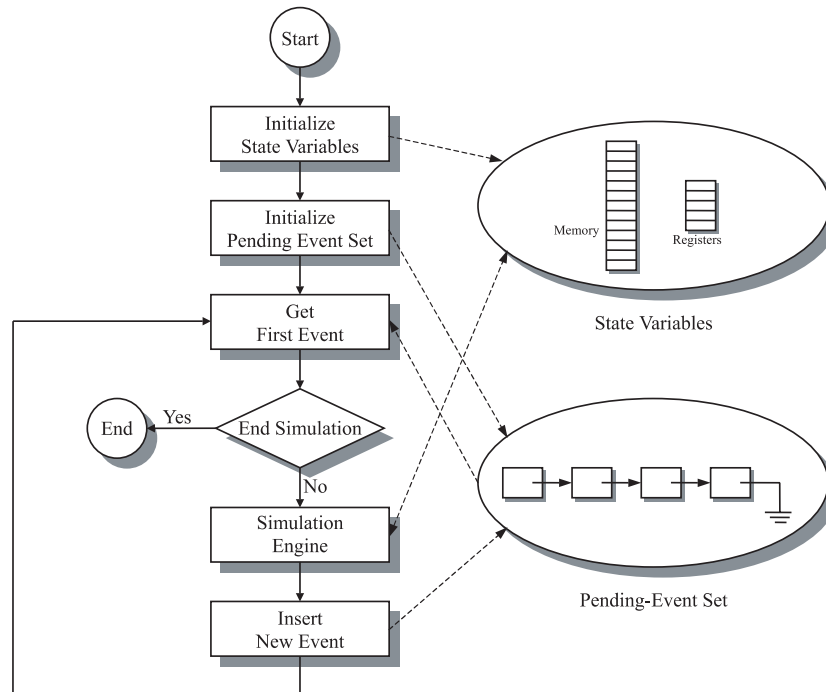


Fig. 3. Discrete-event simulation.

necessary. The simulator continues to do so until the set simulation time is exceeded or no event is left in the pending-event set. A pending-event is a data structure that stores all the events which have not happened and been simulated. The events are recorded sequentially, and it includes its classification and the time when it is recorded.

There are five events: Read Cache, Write Cache, Read Miss Transaction, Write Miss Transaction, and Write Hit Transaction. The CPU will generate a Read Cache event when it reads data in a memory unit. The CPU will generate a Write Cache event when it updates data in a memory unit. When a Read Cache event is simulated, if it is a Read Hit, the data is read and transferred back to the CPU. If it is a Read Miss, a Read Miss Transaction event is generated. When a Write Cache event is simulated, if it is a Write Hit, a Write Hit Transaction event is generated. If it is a Write Miss, a Write Miss Transaction event is generated.

Generally, the way to simulate a multiprocessor is to emulate a time-sharing processing environment which executes a set of “programs” (applications), each consisting of a number of concurrent processes. The scheduling activity is modeled by assigning to each process a time slice, after which it is preempted and the CPU is assigned another process, which is selected from the queue of waiting (ready) processes. The time slice is assumed to be the same for all the processes in the system; no priority is considered, so the processes are all identical. Finally, no synchronization mechanism among the running processes is explicitly modeled.

The key goal in our research is to reduce the total execution time used by the simulator through parallel discrete-event simulation. In practice, the platform is made by connecting a number of workstations through Ethernet, and all the simulators are distributed on all the independent hardware platforms in the network (or on the same hardware platform to execute

concurrent, which is discussed in section 3.1) to do simulation in a parallel architecture. The whole simulation environment can be considered as a distributed computing environment, which will be discussed along with synchronization in the next section.

2.3 Simulation Time and Synchronization

We will use the architectures of two CPUs as an example to demonstrate the process of system simulation and the method of synchronization. As shown in Fig. 4, in the process of simulation, CPU₀ and CPU₁ each has its own variable (a and b) to record the local simulation time while in the memory unit, there is only one variable x to record the system simulation time. When the memory unit receives two events from the CPU, it selects the event with the shorter local simulation time for simulation and it updates the system simulation time to the smaller one. (Assuming that $a < b$, the event with local time a from CPU will be handled first, and the system simulation time will be updated to a).

The simulated event is selected only when every single event from the CPU is collected in a memory unit, so it can be simulated and deleted from the pending-event set. In other words, when one CPU stops sending events to a memory unit, the events from the other CPU will not be handled. To avoid this situation, each CPU will send a null event to keep the system working. Here, we use the logical clock theory of Lamport [24] to prove the accuracy of the synchronization among our simulators. A detailed verification can be found in [24].

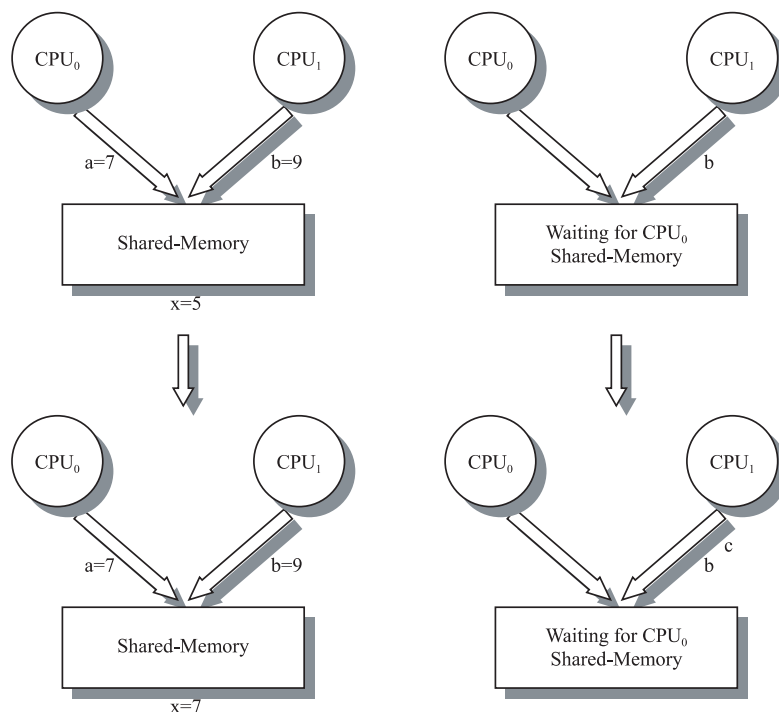


Fig. 4. Simulation time and synchronization.

The logical clock is an abstract point of view in which a clock is just a way of assigning a number to an event, where the number is thought of as the time at which the event occurred. We define a clock C_i for each process P_i as a function which assigns a number $C_i\langle a \rangle$ to any event a in that process. The entire system of clocks is represented by the function C , which assigns to any event b the number $C\langle b \rangle$, where $C\langle b \rangle = C_j\langle b \rangle$ if b is an event in process P_j . The processes need only obey the following implementation rules:

- IR1. Each process P_i increments C_i between any two successive events.
 IR2. (a) If event a is the sending of a message m by process P_i , then the message m contains a timestamp $T_m = C_i\langle a \rangle$.
 (b) Upon receiving a message m , process P_j sets C_j greater than or equal to its present value and greater than T_m .

We define a total ordering relation \Rightarrow as follows: if a is an event in process P_i and b is an event in process P_j , then $a \Rightarrow b$ if and only if either (1) $C_i\langle a \rangle < C_j\langle b \rangle$ or (2) $C_i\langle a \rangle = C_j\langle b \rangle$ and $P_i < P_j$.

2.4 Input/Output Parameters

The simulator is designed to be able to adjust the system architecture and output results through the use of parameters. These parameters are divided into input parameters and output parameters.

Input parameters are used to determine the number of CPUs, the characteristics of the cache, and some relevant time parameters. As shown in Fig. 5, the parameters related to the cache characteristics are the number of bits in cache set, the number of bits for each set with pages, and the number of bits for every page in the cache. As shown in Fig. 6, the time-related parameters are CPU a clock cycle time, the data transfer time between the CPU and the cache, the read and write time for the cache, the data transfer time between caches, the data transfer time between the cache and shared-memory, the read and write time for shared-memory, and the time needed to process addresses from the cache by shared-memory.

Output parameters are used to determine the output result. They are divided into four levels: D1 ~ D4. D1: the simulator only executes the instruction translation without any output. D2: it outputs the memory reference for the application. D3: it outputs the memory

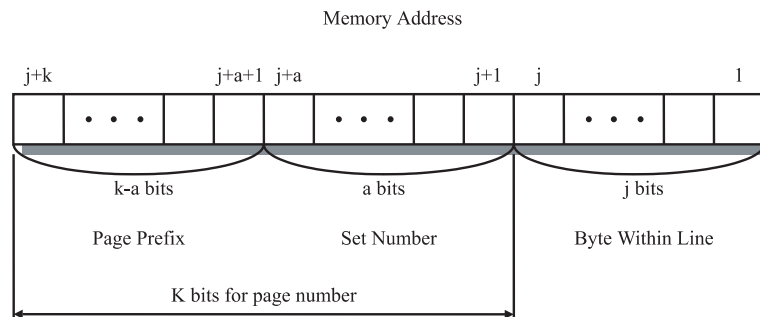


Fig. 5. Cache parameters.

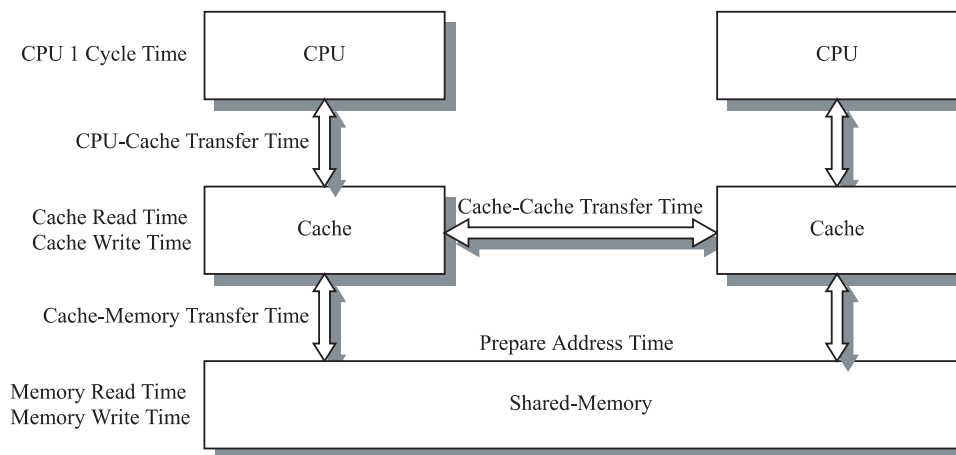


Fig. 6. Time parameters.

reference for the application and the contents of the cache and shared-memory. D4: it outputs the memory reference for the application and the contents of the cache and shared-memory along with all the system events.

3. OVERALL IMPLEMENTATION OF THE SIMULATION ENVIRONMENT

In practice, the simulator consists of two modules, the CPU and a memory unit. The memory unit is divided into three parts: a cache, shared-memory and a cache coherence monitor. These modules are used to simulate the corresponding system elements. The source code was written in the C language and tested on a Sun SPARC 5 workstation with the operating system SunOS 4.1.3.

The simulated CPU was a 80x86 CPU designed by Intel. We studied the operating efficiency of the application program in “multiprocessors/cache/shared-memory.” The CPU was based on the Tanenbaum 8088 simulator [25] and modified for source code to use memory references as the simulator output and also for the purpose of analysis. These memory references were all based on the physical address. We adopted prefetch as the rule for the cache fetch algorithm and used set-associative mapping along with LRU (Least Recent Used) to for the placement and replacement algorithm between the shared-memory and cache.

In sections 3.1 to 3.5, we will discuss the network environment used during simulation, memory simulation, the consistency between the simulator operation architecture and data, multicache coherence, and spinlock design.

3.1 Description of the Simulator and Network Environment

As shown in Fig. 7, four processes are working during simulator operation. (Taking the three-CPU architecture as an example, three CPU processes are used to simulate three CPUs while one Memory process is used for the memory unit). At present, we put the shared-memory, cache, and cache coherence monitor together in one process (to make system to be with higher overhead, which will be discussed in section 4 in detail). During system initiation, the memory process gives every CPU process (CPU_i) a private cache ($Cache_i$). The CPU process communicates with the memory process through the memory access interface (refer to Appendix B.1).

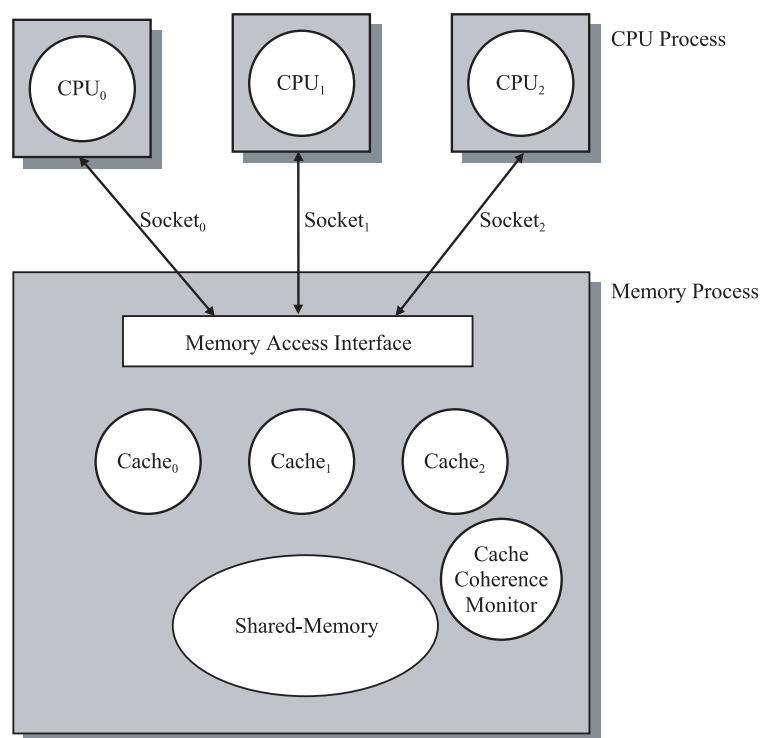


Fig. 7. Schematic structure of the simulator.

As shown in Fig. 8, the network environment used during simulation can be divided into three modes. First is the concurrent mode, in which all the simulators are placed on the same hardware platform and communicate with the memory unit through the UNIX pipe and UNIX domain stream socket. The CPU uses the fork system call from UNIX to simulate the characteristics of the multiprocessor. Second is the parallel mode, in which all the simulators are distributed on different hardware platforms and communicate with the memory unit through the Internet stream socket mechanism of UNIX. Third is the hybrid mode, which allows some simulators to be placed on the same hardware platform and allows some to be distributed on different hardware platforms, so it provides a more convenient and more powerful integrated

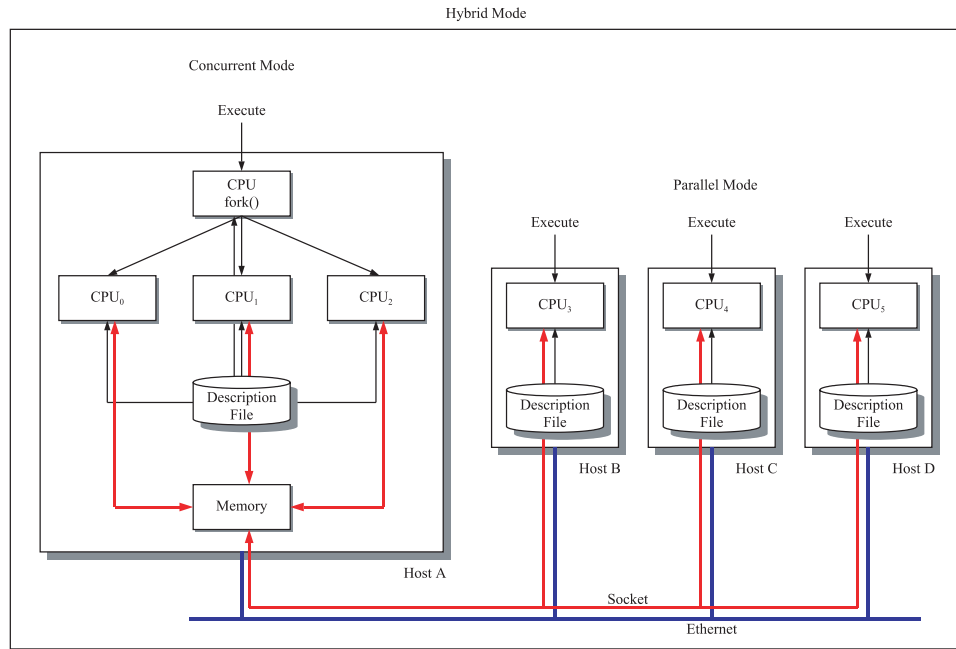


Fig. 8. Schematic structure of the network environment.

environment. When the CPU calls the memory access interface, the memory unit can easily to identify which CPU is sending the request by using the socket identification code, so input/output is carried out correspondingly. The simulator begins by reading the content of the description file and then constructs the system architecture according to the data read. The process also determines the number of system simulation processors, the network address for each CPU and memory unit, and setting connection among each unit.

3.2 Simulation on the Memory Unit

The memory unit contains the cache, shared-memory, and cache coherence monitor. The data structure of the cache is designed by using linked-list and dynamic memory allocation as shown in Fig. 9, which allocates to each CPU its own cache based on the following statement:

```
struct CACHE Cache[CPU_NUMBER];
```

This design enables the cache architecture to be easily adjusted by using parameters (e.g. the number of caches, cache size, number of sets, number of pages in every set and page size) so as to increase the execution efficiency of the cache replacement algorithm. The data structure of the shared-memory uses the following statement to generate a 1MB array (assuming the system has 1MB memory) and to provide data to the cache:

```
unsigned char Memory[1048576];
```

```

struct CACHE_LINE {
    int Tag;          /* Page prefix */
    int Status;      /* Status for cache coherence of this line */
    char *Content;   /* one line has LINE_CONTENT bytes */
    struct CACHE_LINE *NextLine;
};

struct CACHE_SET {
    struct CACHE_LINE *Line; /* One set has LINE_NUMBER lines */
    struct CACHE_SET *NextSet;
};

/* Cache memory has SET_NUMBER * LINE_NUMBER * LINE_CONTENT Bytes */
struct CACHE {
    struct CACHE_SET *Set; /* One cache has SET_NUMBER sets */
};

```

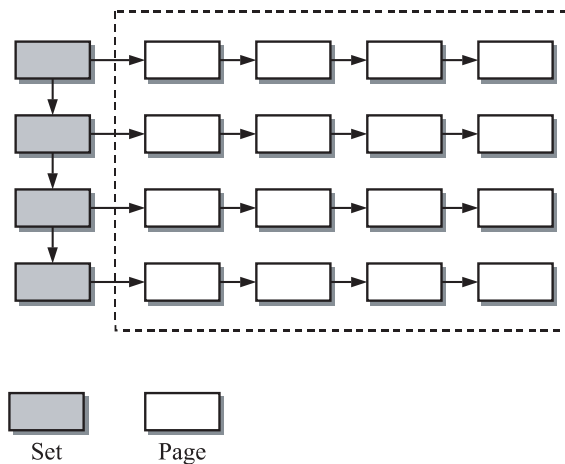


Fig. 9. The structure of the cache.

When the CPU needs to access data in the shared-memory, it calls the memory access interface to get the data in the shared-memory. The memory unit creates a makes memory reference as a performance evaluation record when it gets the operation code, the memory address, and the kind of memory operation (for data/instruction read/write) from the CPU.

The cache corresponds to the shared-memory in the way of prefetch and set-associative mapping algorithm. For example, we divide the shared-memory into 16 sets with 256 pages in each set and 256 bytes in each page while the cache is divided into 16 sets with 4 pages of data in each set. When we assign a shared-memory address X (e.g. address $0xAB478$), the value of bytes at bit 9 to 12 (i.e. $0x4$) can be calculated so as to determine the set number which corresponds to the page containing the shared-memory address (i.e. corresponding to set 4). When the cache needs to take data from the shared-memory, if the cache is full, we use LRU to select a recently unused page in the cache for use as a replacement. Finally, write-back is used to update the data between the cache and shared-memory. We also leave control of the data coherence to the cache coherence monitor (refer to section 3.4).

3.3 The Operation Architecture of the Simulator and Data Coherence

To show clearly the maintaining process for the operation architecture of the simulator and data coherence, we will use the Petri net [26] to describe the whole system and the design of our simulator accordingly. In Fig. 10 shows the operation architecture of the simulator (taking the 3-CPU architecture as an example). In this architecture, there is one token in each of the three <CPU> places, and one token in the <bus free> place is shared. When the CPU performs input/output to and from the cache, the token in the <CPU> place will arrive at the <cache access> place; if there is no need for the bus and the other cache or shared-memory to communicate, then the token returns to the <CPU> place. Otherwise, the token will go to the <get bus> place to check if a token exists in the <bus free> place. If there is no token in the <bus free> place, this means that control of bus has been taken over by the other cache. It does not compete for bus control until a token appears at the <bus free> place (while other caches release bus control). If bus control is taken back, the token in the <get bus> place will go to the <master> place and start to use the bus. In Fig. 11, the Petri net model shows in detail the data coherence when the CPU performs input/output to and from the cache.

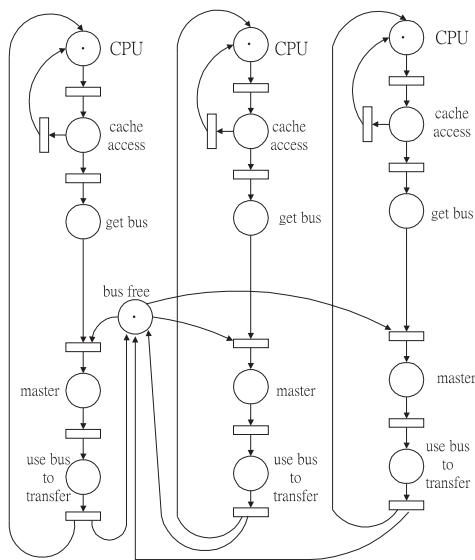


Fig. 10. The operational architecture of the simulator.

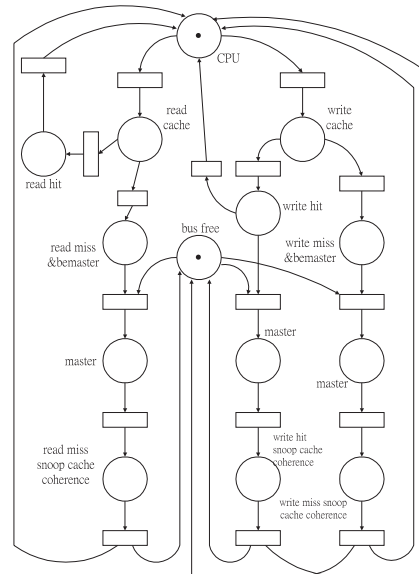


Fig. 11. The operational architecture of data coherence.

3.4 Multicache Coherence

When different CPUs need to do access on the same address at the shared-memory, it is necessary to establish rules to assure data coherence between the cache and shared-memory. In our design, we use the Firefly protocol and IEEE Futurebus⁺ [27] standards to maintain data coherence. Write-through is used to update data among caches while write-back is used to update data between the cache and shared-memory. In addition, the state of every page in the cache is set up as either Shared, Valid Exclusive, and Dirty as shown in Fig. 12. Shared means that besides the cache itself, other shared-memory and at least one cache also has this page of data. Valid Exclusive means that besides the cache itself, other shared-memory also has this page of data with the same content while other caches do not have the data. Dirty means that besides the cache itself, has no other data is newer than this data.

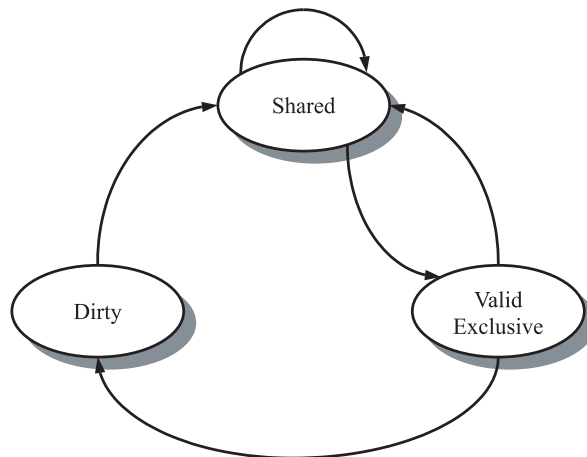


Fig. 12. The state transition diagram of multicache coherence.

When the cache needs to transfer data to another cache, it is necessary to take control of the bus. When more than two caches intend to use the bus, they compete according to the rule of IEEE Futurebus⁺. Every cache sends its number to the bus control line. After a certain time, the result from the signal is kept in the control line through a wire-on operation. In the meantime, every active cache takes this result and checks to see if bus control can be taken back by means of a hardware-defined operation process.

In our bus design standard, every cache needs to oversee bus control maintained by other caches. Because the two events Read Cache and Write Cache are generated by the CPU and sent to the memory unit, they have nothing to do with bus operation. The three events Read Miss Transaction, Write Miss Transaction, and Write Hit Transaction correspond to three transactions, so in real simulation, we use method way of direct notice to the cache to simulate how the cache oversees the bus; i.e. every cache is instructed to respond to this event according to the cache number sequence (which is also the sequence used to compete for bus control). The response includes data coherence control and transition between states.

3.5 Simulation of Spinlock

Our simulator uses the spinlock function to handle the mutual exclusion problem, which is faced upon for the application program under parallel environment. We design the test-and-test hardware simulation in shared-memory. At the beginning, the address for the lock variable is sent to the shared-memory, and then the value of the lock variable is read out and set to 1. The above operation is completed in one clock cycle. We add three instructions, the test-and-set lock variable, load lock variable, and unlock lock variable, to the CPU. The test-and-set lock variable instruction reads the value of the lock variable into the AL register and sets the value of the lock variable to 1 at the same time. The load lock variable instruction reads the value of the lock variable in the AL register. The unlock lock variable instruction sets the value of the lock variable to 0.

Fig. 13 shows how the three instructions, test-and-set lock variable (TAS), load lock variable (LL), and unlock lock variable (UL), are used to achieve spinlock. When the process enters a critical section, it first executes the “LL AL, La” instruction to read the value of La

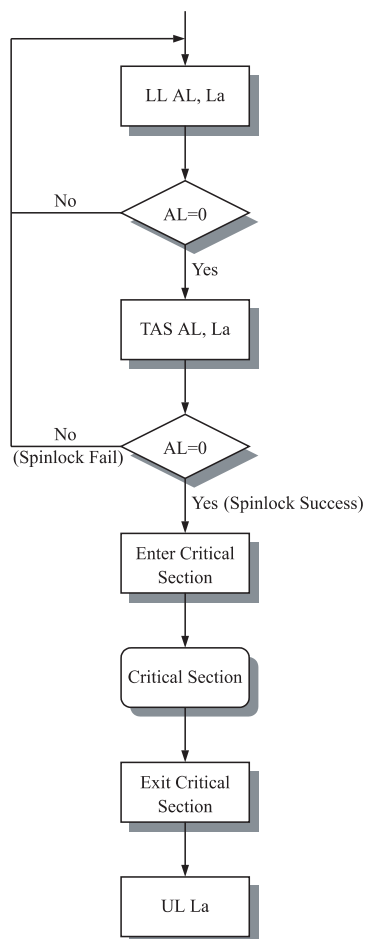


Fig. 13. Simulation of spinlock.

(assuming that L_a is a lock variable) into the AL register; then it checks if the value of AL is 0. If the value of AL is not 0, this means that the present state is the lock state. It then continues to read the value of L_a until it is 0. If the value of AL is 0, it executes the instruction “TAS AL, L_a ” and checks if L_a is 0. If it is not 0, the process returns to the beginning of the cycle to restart. If the value of AL is 0, this means that spinlock is successful, and the process is allowed to enter the critical section. When the process leaves the critical section, it executes the “UL L_a ” instruction to set the L_a value to 0.

4. SIMULATION AND RESULTS

In this section, we will present simulation experiments conducted using some multi-processor trace data. In section 4.1, we will review the previous results obtained using the efficient cache memory evaluation technique. In section 4.2, the characteristics of the trace data will be given. Section 4.3 will present the simulation results.

4.1 Stack Evaluation

A cache block replacement algorithm is called a stack algorithm if, when it is used, the cache contents in a two-level hierarchy always satisfy an inclusion property; that is, the contents of a smaller cache is always a subset of those of a large cache for any memory access sequence by the LRU [28]. Several popular replacement algorithms, including LRU, least-frequently-used (LFU), and minimal (MIN, which replaces replacing the block that won't be accessed till the farthest future), are stack algorithms. For stack replacement algorithms, there exists a one-pass procedure, called stack evaluation, that produces hit ratios for the entire range of cache sizes with one pass over any access trace. As explained below, during one-pass processing of a trace, data on the stack distance frequencies about the use of data blocks are collected, and hit ratios are later derived from these data. A detailed explanation of and proof for stack evaluation can be found in [28].

In practice, we use parallel discrete-event simulation to get memory references. Then by using the memory references with stack evaluation, evaluation can be performed on the efficiency of the cache architecture with different parameters.

4.2 Trace Data

Three traces of parallel applications were used: FFT, Wavelet Transform, and Parallel Wavelet Transform. They were generated by means of software using our simulator and executed on a SPARC 5. FFT is a radix-2 fast Fourier transform application. Wavelet Transform and Parallel Wavelet Transform applications are transformed for seven levels by using the coefficient of the Dabuechies filter. The three applications all used the standard image “Lena” with 256x256 resolution and 256 gray levels to perform the test. Each reference record consisted of a one-byte CPU number (socket file descriptor), a one-byte operation code, a 32-bit address space (but only used 20-bit), a one-byte memory operation (for data/instruction read/write), and a one-byte spinlock. Data and instruction accesses were “unified”: they were treated as being cached together.

4.3 Simulation Results

Fig. 14 shows the simulation results for FFT, Wavelet Transform, and Parallel Wavelet Transform. At the present time, the cache replacement algorithm that our simulator uses is LRU. Other replacement algorithms are also available (e.g. LFU and MIN). We will continue to add these replacement algorithms to our simulator as options for users, which will make our simulator more complete.

Fig. 15 shows the execution time on substantial machine and the simulation time used by the simulator on different hardware platforms. Fig. 16 shows a comparison between our simulator and other simulators. Except for MINT, which simulates shared-memory multiprocessors, they all simulate a single processor. MINT also simulates multiprocessors in a sequential way. We provide a shared-memory multiprocessor simulator that can execute 80x86 programs. On the other hand, we reduce the simulation time through parallel discrete-event simulation. The slowdown is from 20 to about 800 (depending on output parameters and it has higher overhead).

5. CONCLUSIONS AND FURTHER RESEARCH

In this section we will present conclusions and further research.

5.1 Conclusions

In our research, we have implemented a shared-memory multiprocessor simulator that can execute 80x86 programs. The simulator can execute programs through a network so as to cooperate with other workstations to solve a problem. The simulator can adjust the system architecture by using parameters according to the user's needs, such as the number of CPUs, the characteristics of the cache, and relevant time parameters. Besides this, the simulator can conduct program debugging and calculation of the simulation time and the number of memory references. By analyzing the memory references for the application program, the simulator can find the parameter with higher cache hit ratio so as to improve the efficiency for the next execution.

In practice, we provide technology which can simulate a virtual computer system and also can evaluate the efficiency of our desired system. These can be used as references for adopting or modifying a system so as to increase production efficiency and reduce R&D costs. Moreover, we provide a parallel processing environment for execution of application programs.

The user interface needs to be improved. When the simulator is working, the system architecture configuration relies on manual operation. We hope we will be able to provide an integrated environment in which only some relevant information will be needed for the system to configure the program execution environment. At present, our simulator places the cache and shared-memory in the same process. Each memory input/output needs to execute network communication. This makes the system overhead too high, resulting in poor simulator efficiency. We hope each module in the simulator can be divided into parallel operation modules to increase the simulator efficiency. Now, we can only simulate the hardware architecture of a multiprocessor system. In fact, there is a lack of control when an application program is executed in a parallel processing environment. In future research, we

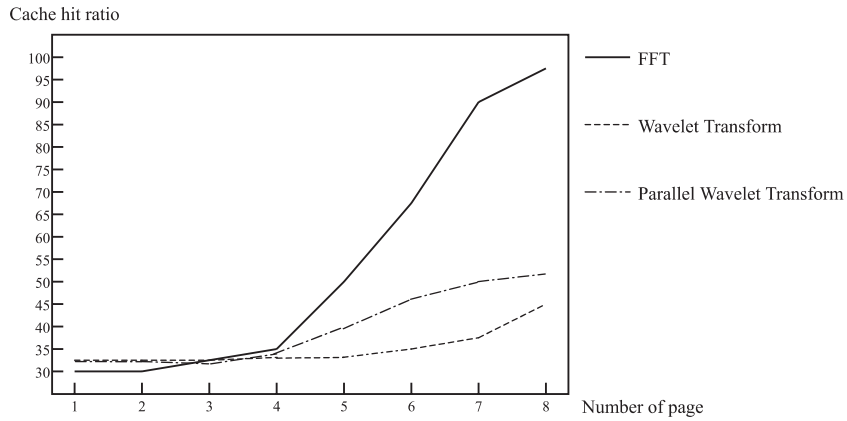


Fig. 14. Cache hit ratios for FFT, Wavelet Transform, and Parallel Wavelet Transform.

	FFT	Wavelet Transform	Parallel Wavelet Transform
Pentium 90 Substantial	7.31	18.11	3.07
Sun SPARC 5 Simulation	1893.29	4762.93	803.73
IBM RS6000 Simulation	916.89	2332.02	396.03
Pentium II 233 Simulation	528.73	1290.34	208.70
Run-time in seconds			

Fig. 15. Real execution time and simulation times on different platforms.

Reference	Name	Target(s)	Host(s)	Slowdown
CmelikandKeppel1993	Spa(Spy)	SPARC	SPARC	40-600
Davies 1994	Mable	MIPS-I,MIPS-III	MIPS-I	20-200
Lzrus 1991	SPIM	MIPS-I	SPARC,680x0,MIPS, x86,HP-PA	25
VeonstraandFowler 1994	MINT	R3000	R3000	20-70
CmelikandKeppel1994	Shade	SPARC-V8,SPARC-	SPARC-V8 V8,MIPS	9-14
Chen1998	86sin	80x86	80x86	20-800

Fig. 16. Comparison between our simulator and other simulators.

hope to add some control in this regard so that we can evaluate the efficiency of an application program using the simulator. Finally, we hope to implement a good parallel processing environment in a network.

5.2 Further Research - Letni Project

We call our new project Letni. In this project, the original 80x86 simulator will be rewritten in the C++ and Java languages to support more hardware platforms, such as IBM RS6000/AIX, Sun Ultra 1/ Solaris Intel Pentium III/Linux etc. The CPU instruction set will be simulated and expanded to get an 80486 instruction set, and will directly support memory management, protection, and multitasking for 80486 in the 32 bit protection mode in order to enhance the capabilities of simulator [29].

The Letni project will combine the CORBA system experimentally. We will try to distribute all the simulator components in the network in the way of Java applet by using the transparency to hardware platform from Java virtual machine and the transparency to network from the CORBA system. We will directly use the Internet as our network environment-development platform. The huge network resources can provide powerful calculation ability.

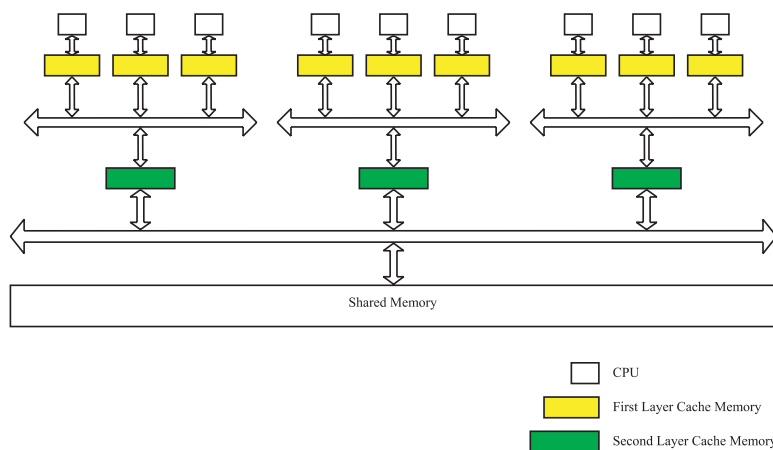


Fig. 17. The architecture of a multiprocessor and multicache machine.

Aside from this, we hope to continue to develop the multilevel cache architecture under the single bus architecture between the CPU and shared-memory, as shown in Fig. 17, and we will also use the IEEE Futurebus⁺ standard to simulate the bus and cache coherence. Further, we will provide more coherence protocol simulation elements from which to choose. As for the shared-memory, actually, it puts memory addresses in different ICs to increase the operation efficiency of the shared-memory. This technology can be applied to the cache, which will be a focus of our future research.

APPENDIX

A. Trace-Driven Simulation

Trace-driven simulation writes a program that simulates the behavior of a proposed memory system design and then applies a sequence of memory references to the simulation model to mimic the way that a real processor might exercise the design. When a memory reference is derived and saved as a file, there is no need for other special hardware to repeat and to produce different kinds of simulation results.

A.1 General Evaluation Criteria and Metrics

In this paper, the criteria we use to evaluate trace-driven simulation are as follows:

1. Slowdown: The simulator is implemented on different hardware platforms, so it has different processing rates which are hard to compare. Slowdown is adopted to show the corresponding relationship between the host hardware and simulator:

$$\text{Slowdown} = \frac{\text{Total Simulation Time}}{\text{Normal Host System Execution Time}}.$$

2. Memory usage: Different hosts have different processing rates between primary and second storage, so memory overhead is used for the purpose of evaluation:

$$\text{Memory Overhead} = \frac{\text{Additional Memory Required}}{\text{Normal Host Memory Required}}.$$

3. Portability: The ease to re-implement to different hardware platforms.
4. Flexibility: The ease of simulating a wide range of memory parameters (cache size, line size, associativity, replacement policy, etc.) and collecting a broad range of performance metrics (miss ratio, misses per instruction, cycles per instruction, etc.).
5. Expense: If the cost of hardware or special monitoring equipment is high.
6. Ease of use: How easily end-user learns and operates the trace-driven simulator.

B. Instruction-Set Emulation

The core program is based on an endless loop. Using the C language switch statement, it performs instruction translation (refer to Appendix B) according to every operation code in an 80x86 processor. It saves a record of the memory reference by means of the input/output process:

```
while (True) {
    switch (op) {
        case 0xA4:
            Memory Access (Socket FD, &t_for_rep, (int)xapc, READ,0);
            BSTORE(t_for_rep);
            BSIDI;
            return;
        case 0xA5:
            Memory Access (Socket FD, &t_for_rep, (int)xapc++,READ,0);
            BSTORE(t_for_rep);
```

```

    eapc++;
    Memory Access (Socket FD, &t_for_rep, (int)xapc, READ, 0);
    BSTORE (t_for_rep);
    WSIDI;
    return;
    :
}
}

```

B.1 Memory Access Interface

The MemoryAccess is the memory access interface, the prototype of which is as follows:

```

void Memory Access (Socket FD, Opcode, Address, Operation, Lock)
char Socket FD;           /* Internet stream socket file descriptor */
unsigned char *Opcode;    /* Operation code */
char Address;             /* The shared-memory address (20 bits) */
char Operation;          /* READ or WRITE memory operation */
char Lock;                /* Lock mechanism (test-and-set) */

```

REFERENCES

1. P. Stenstrom, "A survey of cache coherence schemes for multiprocessors," *Computer*, Vol. 23, No. 6, 1990, pp. 12-25.
2. M. Heinrich, M. Rosenblum, J. Hennessy, etc., "The performance impact of flexibility in the Stanford FLASH multiprocessor," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994, pp. 274-284.
3. J. R. Goodman, "Cache memory optimization to reduce processor-memory traffic," *Journal of VLSI and Computer Systems*, Vol. 2, No. 1-2, 1987, pp. 61-86.
4. A. J. Smith, "Cache memories," *Computing Surveys*, Vol. 14, No. 3, 1982, pp. 473-529.
5. J. Archibald and J. L. Baer, "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Transactions on Computer Systems*, Vol. 4, No. 4, 1986, pp. 273-298.
6. M. Dubois and F. A. Briggs, "Effects of cache coherency in multiprocessors," *IEEE Transactions on Computers*, Vol. 31, No. 11, 1982, pp. 1083-1099.
7. S. J. Eggers, "Simulation analysis of data sharing in shared memory multiprocessors," PhD dissertation, Department of Computer Science and Engineering, University of California, Berkeley, 1989.
8. R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins and R. G. Sheldon, "Implementing a cache consistency protocol," in *Proceedings of 12th Anniversary International Symposium on Computer Architecture*, 1985, pp. 276-283.
9. M. Tomasevic and V. Milutinovic, eds., *The Cache Coherence Problem in Shared-Memory Multiprocessors-Hardware Solutions*, Los Alamitos, Calif.: IEEE CS Press, 1993.

10. Q. Yang, L. N. Bhuyan and B. C. Liu, "Analysis and comparison of cache coherence protocols for a packet-switched multiprocessor," *IEEE Transactions on Computers*, Vol. 38, No. 8, 1989, pp. 1143-1153.
11. D. A. Reed and R. M. Fujimoto, *Multicomputer Networks Message-Based Parallel Processing*, The MIT Press, Cambridge, Massachusetts, London, England, 1987.
12. M. Ajmone-Marsan, G. Balbo and G. Conte, *Performance Models of Multiprocessor Systems*, The MIT Press, Cambridge, Massachusetts, London, England, 1986.
13. J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu and A. Gupta, "Hive: Fault containment for shared-memory multiprocessors," *The 15th Symposium on Operating Systems Principles*, 1995, pp. 12-25.
14. M. Rosenblum, S. A. Herrod, E. Witchel and A. Gupta, "Fast and accurate multiprocessor simulation: the SimOS approach," *IEEE Parallel and Distributed Technology*, Vol. 3, No. 4, 1995, pp. 34-43.
15. J. Misra, "Distributed discrete-event simulation," *ACM Computing Survey*, Vol. 18, No. 1, 1986, pp. 39-65.
16. C. C. Chou, "Parallel simulation and its performance evaluation," Technical Report 93-02, PhD dissertation, Department of Computer Science, the University of Iowa, Iowa City, USA, 1993.
17. E. Witchel and M. Rosenblum, "Embra: faster flexible machine simulation," in *Proceedings of SIGMETRICS'96*, 1996, pp. 68-79.
18. R. A. Uhlig and T. N. Mudge, "Trace-Driven Memory Simulation: A Survey," *ACM Computing Surveys*, Vol. 29, No. 2, 1997, pp. 128-170.
19. R. Cmelik and D. Keppel, "Shade: A fast instruction-set emulator for execution profiling," Technical Report UWCSE 93-06-06, Department of Computer Science and Engineering, University of Washington, 1993.
20. P. Davies, P. Lacroute, J. Heinlein, and M. Horowitz, "Mable: A technique for efficient machine simulation," Technical Report CSL-TR-94-636, Department of Computer Science and Engineering, Stanford University, 1994.
21. J. R. Larus, "SPIM S20: A MIPS R2000 simulator," Department of Computer Science, University of Wisconsin-Madison, Technical Report, Revision 9.
22. J. Veenstra and R. Fowler, "MINT: A front end for efficient simulation of shared-memory multiprocessors," in *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications System (MASCOTS)*, 1994, pp. 201-207.
23. B. Cmelik and D. Keppel, "Shade: A fast instruction-set emulator for execution profiling," *Proceedings of the 1994 SIGMETRICS Conference on Measurement and Modeling of Computer Systems (Nashville, TN)*, ACM, New York, 1994, pp. 128-137.
24. L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, Vol. 21, No. 7, 1978, pp. 558-565.
25. From <ftp://netbsd.csie.nctu.edu.tw/pub/Minix/simulator>
26. S. S. Muchnick, *A Primer in Petri Net Design*, SUN Microsystems, Inc.
27. Institute of Electronics and Electronics Engineers, Inc., "IEEE standard for Futurebus⁺ - logical protocol specification," IEEE Std896.1, 1994.
28. R. Mattson, J. Gecsei, D. Slutz and I. Traiger, "Hierarchical storage evaluation techniques," *IBM Systems Journal*, Vol. 17, No. 2, 1970, pp. 78-117.
29. P. Brumm, D. Brumm and L. J. Scanlon, *i486 Microprocessor Programmer's Reference Manual*, Intel Corporation, 1990.



Po-Zung Chen (陳伯榮) received the Ph.D. degree in Computer Science from the University of Iowa in December 1989. From November 1989 to May 1990, he was a visiting Assistant Professor at Michigan Technological University (Houghton, Michigan). Since August 1990, he is an Associate Professor with the Department of Computer Science and Information Engineering at Tamkang University (Taipei, Taiwan). His research interests include object-oriented distributed programming, parallel & distributed systems, and simulation & modeling.



Shyh-Nong Chen (陳士農) received his B.S. degree in Computer Science and Information Engineering from Tamkang University (Taipei, Taiwan) in June 1995. He is currently working toward his Ph.D. degree in Computer Science and Information Engineering at Tamkang University. His research interests include image processing, object-oriented distributed programming, parallel & distributed systems, and simulation & modeling.