*Proc. Natl. Sci. Counc. ROC(A)*
Vol. 22, No. 4, 1998. pp. 524-538

# SEESMA: A Simulation and Evaluation Environment for Shared-Memory Multiprocessor Architecture

CHAO-CHIN WU, DER-LIN PEAN, JEN-PIN SU, JIA-RONG WU, HSUAN-WOEI HUANG, JUN-LONG HUANG, JIA-LIN LEE, HUEY-TING CHUA, AND CHENG CHEN†

*Department of Computer Science and Information Engineering*
*National Chiao Tung University*
*Hsinchu, Taiwan, R.O.C.*

(Received October 9, 1997; Accepted January 14, 1998)

**ABSTRACT**

We have constructed a simulation environment called SEESMA (a Simulation and Evaluation Environment for Shared-memory Multiprocessor Architecture). It is a program-driven simulator consisting of a memory reference generator and a target system simulator. SEESMA aids in the investigation of shared-memory multiprocessor architecture through a user friendly interface. It is a software platform with a modular structure suitable for educational and research purposes. To achieve the design objectives, SEESMA supports the following simulation environment for various important design spaces: (1) two types of processing element architectures; (2) a two-level cache structure with write cache; (3) five types of memory consistency models; (4) five types of cache coherence protocols; (5) three types of directory structures; (6) interconnection networks, and other related issues. Each sub-environment provides many options for investigating the interactions between different options. Meanwhile, we have also developed an X-window interface to specify system architectures and benchmarks in addition to friendly on-line help. We can determine the setup values of architectural parameters in a status region by moving the cursor around the graph of each architecture component. The architecture graph is changed each time the user modifies the simulation options. In addition, SEESMA can automatically collect data from several simulation results and display a comparison using either a bar graph or a curve graph. So far, many design issues and important performance gains for high performance multiprocessor systems can be evaluated and explored using this platform.

**Key Words:** shared memory, multiprocessor, simulator, program-driven simulation, system design, performance evaluation, parallel-multithreaded processor

## I. Introduction

Memory subsystem researchers working on multiprocessor systems usually rely on a simulator to evaluate and verify their designs. Because system components can potentially interact in every memory reference operation, the simulation of multiprocessor systems is computationally expensive. Three sequential simulation techniques have been proposed for low-cost workstations: trace-driven simulation (Eggers *et al.*, 1990), execution-driven simulation (Davis *et al.*, 1991), and program-driven simulation (Veenstra and Fowler, 1994).

Trace-driven simulation does not execute real benchmark programs at simulation time. Instead, it uses memory reference traces as its input. Though such a simulation method is very efficient, it suffers from several disadvantages: (1) the timing or interleaving of memory references can not be changed because there is no feedback from the simulator to the trace file during simulation; (2) the trace may be not accurate because the timing difference between the simulation time and the collection time can cause programs to take different branches; and (3) each trace file can require hundreds of megabytes of disk space (Eggers *et al.*, 1990). In contrast, execution-driven simulation and program-driven simulation execute real benchmark programs during simulation. Their advantage is that they provide accurate reference timing and execution paths. Execution-driven simulations modify benchmark programs by inserting calls into the simulator source. However, this technique has the following disadvantages. (1)

†To whom all correspondence should be addressed.

Users have to modify all the benchmark programs which are to be simulated. (2) We have to relink the simulator with the benchmark program each time the simulator is modified. (3) Memory references for library code called by the user program cannot be generated, unless the library source is available and statically linked to the user program (Davis *et al.*, 1991).

Program-driven simulation overcomes the above problems. It requires that benchmark programs be statically linked with library codes to be executable files for a specific real machine. These statically-linked executable files are used as input files for program-driven simulators. Consequently, benchmark programs require no additional modification or relinking. In addition, the memory references for the library code can be simulated even if a library source is not available (Veenstra and Fowler, 1994).

MINT (MIPS interpreter) is a kernel of program-driven simulators (Veenstra and Fowler, 1994). We have extended MINT to construct a simulation and evaluation environment named SEESMA (a Simulation and Evaluation Environment for Shared-memory Multiprocessor Architecture). SEESMA aims to provide a simulation and evaluation environment for shared-memory multiprocessor systems. It supports the following simulation functions: (1) two types of processor architecture, (2) a two-level cache with write caches; (3) a message-passing based interconnection network; (4) five types of memory consistency models; (5) five types of cache coherence protocols; and (6) three types of cache coherence directory structures. SEESMA has versatile simulation options to choose from. Users can customize their simulated target memory architecture by switching individual options on or off through a friendly X-window interface.

We provide both menu-based and graph-based input interfaces. The latter shows an architecture graph, which changes each time the user modifies a simulation option. Moreover, we can determine the setup values of the architectural parameters in a status region by moving the cursor around each architecture component. During the setup procedure for the parameters, the graphical interface also supports on-line help for users. In addition, SEESMA can automatically collect the data from several simulation results and display a statistical comparison using a graph. So far, both bar graphs and curve graphs are available. By means of the graphs in the integrated environment, users can compare different architectural parameters more easily.

SEESMA provides so many features that it is a good research environment for those who are interested in system designs for shared-memory multiprocessor architectures. We have used SEESMA to explore the characteristics of cache coherence protocols and write caches, and to verify our proposed methods either for architecture designs or for compiler designs.

The rest of this paper is organized as follows. Section II describes the simulated system architecture and gives an overview of SEESMA. Section III details the simulation functions of SEESMA. In Section IV, we present the graphic input/output interface, which facilitates use of our simulation and evaluation environment. Section V gives an evaluation example based on six SPLASH (Stanford Parallel Applications for Shared-Memory) benchmark programs run on SEESMA. Finally, we conclude the paper and suggest future development.

# II. Overview of SEESMA

SEESMA is a sequential program that simulates the execution of a benchmark program on any number of processors. It is an extension of the MINT package (Veenstra and Fowler, 1994). Its execution model is the same as that of the MINT package; therefore, it is a program-driven simulator. MINT itself only simulates the ideal parallel memory that can satisfy a request from every processor in every cycle, even if multiple references are to the same location. Under this memory model, all memory and synchronization references complete in a single cycle. Therefore, to simulate more realistic memory systems, SEESMA integrates the MINT package with our constructed memory system simulator and interconnection network simulator.

In the following subsections, we will give an overview of the MINT and the organization and construction of SEESMA, and finally, discuss the correctness of simulation runs on SEESMA.

## 1. The MINT

MINT is a program-driven simulator for the study of multiprocessor systems (Veenstra and Fowler, 1994). It consists of a memory reference generator, simulation library, and programming environment (Veenstra and Fowler, 1994). The reference generator interprets MIPS codes and models the execution of an application program on any number of processors. The simulation library manages and schedules events and processes. These two parts are together called the *front end*, and they are a source of overhead from the perspective of a memory system researcher. MINT supplies the front end to enable designers to focus on the design of memory systems and interconnects.

The programming environment provides programmers with a set of function calls initialized empty. When an event (for instance, a read reference) occurs,

the memory reference generator calls the appropriate functions (in this case, sim_read()) to simulate the event. Because the function calls are initialized empty, the memory system simulated by the MINT package is an ideal model. In other words, the memory system has neither cache hierarchy nor memory access latency. What users have to do to simulate their target memory systems is to replace the original function calls with new contents. In addition, users may define other function calls if this is needed. The content of each user-defined function call is programmed individually with the appropriate behavior, the local timing control routine, and, if necessary, the time point at which to call other functions. These function calls are together called the *back end.*

The front end implicitly maintains the global time by means of a data structure, called the Time_Wheel. The Time-Wheel is a circular array in which each element points to a prioritized task queue for a certain time point. When the global time advances to the time point, the tasks queued in the element are processed one by one from the head to the tail of the queue. The global time is not increased until we have processed all the tasks queued in the array element pointed to by the current time.

A task is processed by calling its associated function. The task may create other new tasks to continue the rest of its sub-tasks. For example, a write reference will issue several invalidation requests to other caches after it finds that the block status is shared. According to their timings and priorities, these new tasks are queued on the Time_Wheel. If a function returns a T_ADVANCE value, the associated thread continues its execution until an interesting operation (typically the generation of a memory reference) is encountered. At this time, the front end will call the appropriate back-end function. However, if a T_FREE value is returned, the front end just frees the task and processes the next task.

If several tasks ask to access the same hardware resource and the hardware resource can service only one request at a time, only the event processed first is granted service. The others are scheduled with new events for retry at a later time. These tasks take place and are handled virtually in parallel.

## 2. Organization and Construction of SEESMA

The primary goals of SEESMA are to use the MINT-support programming environment to construct simulations of memory subsystems and interconnection networks of shared-memory multiprocessor systems. SEESMA is comprised of the MINT front end (Veenstra and Fowler, 1994) and our constructed target

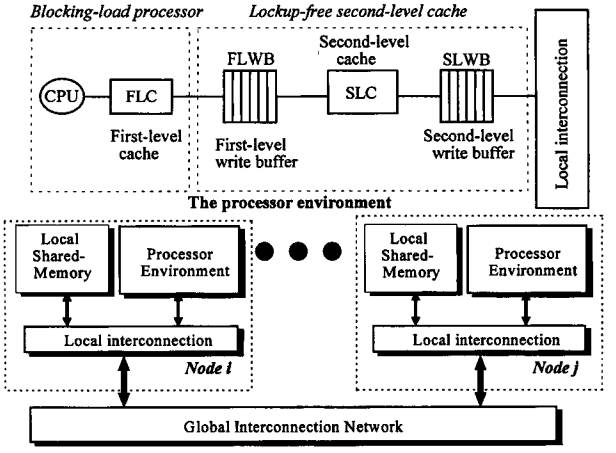


**Fig. 1.** Overview of the general simulated multiprocessor architecture.

memory system and interconnection network simulators (the back end).

The general architecture SEESMA simulates is a distributed shared-memory multiprocessor system as shown in Fig. 1. It is composed of multiple nodes connected by a global interconnection network. Each node is composed of a processor environment, a local shared memory, and a local interconnection. The organization of the processor environment consists of a CPU and a two-level cache. In our two-level cache hierarchy, the first-level cache (FLC) is a write-through cache with no allocation on write misses, and the second-level cache (SLC) is a write-back cache with write allocation on write misses. The cache system is blocked on read misses but not on write misses. Between the FLC and the SLC there is a first-level write buffer (FLWB) to avoid processor stalls on write accesses. Moreover, to make the SLC lock-free, meaning that multiple outstanding write requests are allowed, a second level write buffer (SLWB) is included to store all writes to the SLC that cause global actions. The cache hierarchy is interfaced to the local portion of the shared memory and the global interconnection network. The shared memory is divided into several equal-size pages that are allocated to nodes in a round-robin fashion.

SEESMA has been programmed in C language with about 22000 statements and constructed on a SUN workstation running under the UNIX System V environment. The general back-end structure of SEESMA as shown in Fig. 2 is similar to the architecture illustrated in Fig. 1. The whole structure is modular in design so that it can be enhanced or extended easily. However, SEESMA provides so many simulations of various architectural functions that we can define an architecture with specified components. The primary simulation functions include the following:

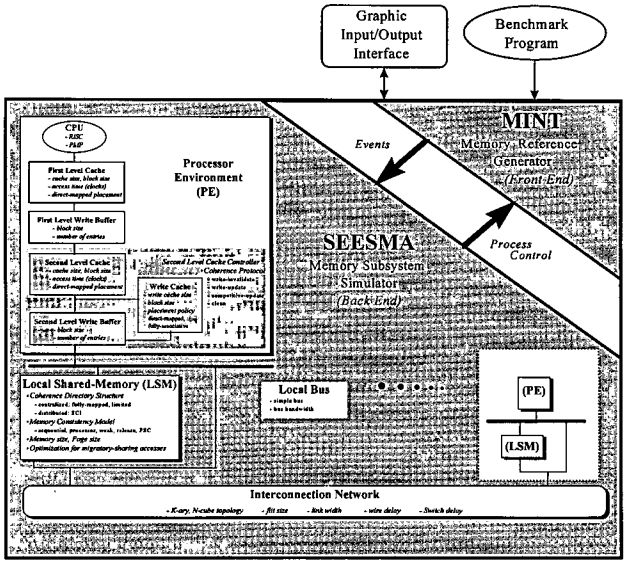


**Fig. 2.** Overview of SEESMA.

(1) Two CPU types: RISC (Reduced Instruction Set Computer) and PMP (Parallel Multithreaded Processor) (Hirata *et al.*, 1992).
(2) Five memory consistency models: sequential (Lamport, 1979), processor (Goodman, 1989), weak (Dubois *et al.*, 1986), release (Gharachorloo *et al.*, 1990), and our proposed PSC (PMP-MP Specific Consistency) models (Wu and Chen, 1998b).
(3) Two-level cache hierarchy with FLWB, SLWB, and write cache (Dahlgren and Stenstrom, 1995).
(4) Three cache directory structures: centralized fully-mapped, centralized limited (Stenstrom, 1990), and distributed SCI (Scalable Coherence Interface) structures (Gjessing *et al.*, 1991).
(5) Five cache coherence protocols: SCI, write invalidate, write update, competitive-update, and clean protocols (Grahn *et al.*, 1995).
(6) K-ary, N-cube interconnection network (Dally, 1990).
(7) Optimization for migratory-sharing accesses (Su *et al.*, 1996).

So far, we provide almost any combination of options of these seven primary functions except for the following combinations: (1) the clean protocol or the PSC model under the distributed directory structure; and (2) the SCI protocol or the optimization algorithm for migratory-sharing accesses under a centralized directory structure.

After constructing the back end, we can compile it on a SUN workstation. We can then link it with the compiled memory reference generator and libraries to obtain an executable simulator as shown in Fig. 3. Now, we can use SEESMA to evaluate various benchmark programs, which are statically-linked Irix executable file compiled for the MIPS R3000 processor. Meanwhile, we have to setup simulation options to customize a simulated target machine. Basically, SEESMA can output the following important simulation and evaluation results:

(1) Parallel execution time: including busy time, stalled time for read miss, acquire-stall time, stalled time for write-buffer-full, and contention time for accessing FLC.
(2) Memory accesses for FLC and SLC: including any number of read and write requests.
(3) Miss ratio for read and write requests.
(4) Cold and coherence misses.
(5) Distribution of the invalidation/update count.
(6) Write run distribution.
(7) Hit ratio in write cache.
(8) Write run distribution in write cache.
(9) Number of read and write misses to invalid memory copies.
(10) Amount of network traffic.
(11) Number of LOCK and Barrier operations.
(12) Delay time for acquire and release accesses.

## 3. Correctness

Correctness is the most important issue for any software design. For a small to medium input domain, it is easy or possible to verify the correctness. However, for a large input domain, correctness verification is difficult or even impossible. What we can do is to choose several reported results, redo the simulation using SEESMA, and finally compare these two results. For instance, Dahlgren and Stenstrom (1995) have

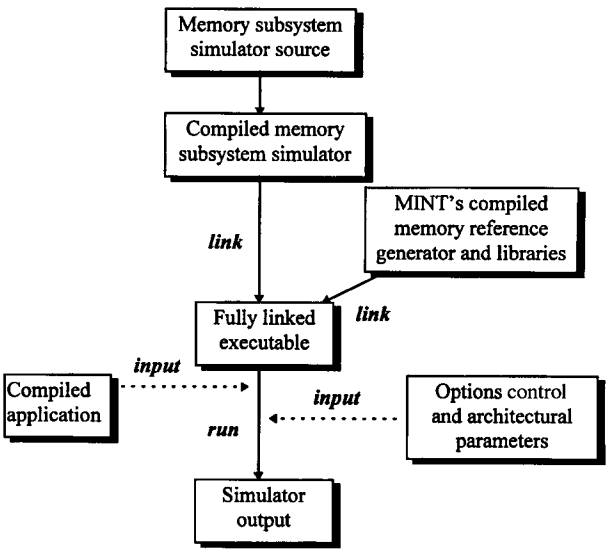


**Fig. 3.** Construction of a MINT-based simulator.

**Table 1.** Mapping Table for Memory Accesses under Different Memory Models

| Type \ Model | SC | PC | WC | RC |
|---|---|---|---|---|
| Ordinary Read | Read | Read | Read | Ordinary Read |
| Ordinary Write | Write | Write | Write | Ordinary Write |
| Acquire | Read | Read | Synchronization | Acquire |
| Release | Write | Write | Synchronization | Release |

studied the impact of write caches on several cache coherence protocols for shared-memory multiprocessors. On the other hand, we have also studied this topic for Parallel multithreaded processor-based multiprocessor (PMP-MP) systems (Wu and Chen, 1998a). In this research paper, we provide evaluation results for PMP-MP. The PMP-MP architecture with one thread per processing element is the same as that simulated by Dahlgren and Stenstrom. Comparing these two results, we can see that they have the similar behaviors. The only exception is for the Ocean program. This is because the Ocean program Dahlgren and Stenstrom used is in the SPLASH suite (Singh *et al.*, 1992) that which we have used is in the SPLASH-2 suite (Woo *et al.*, 1995).

In the following section, we present details of the functions and structure of SEESMA.

# III. Functions of SEESMA

The memory system we simulate in the back end of SEESMA is a cache-coherent non-uniform memory architecture (CC-NUMA). The back end consists of simulations of a two-level cache, global interconnection network, memory consistency model, and cache coherence protocol. Details of each simulation are given in the subsections below.

## 1. Multithreaded Multiprocessors

PMPs (Hirata *et al.*, 1992) can execute more than one thread at the same time. We expect that this type of processor will become one of the most popular single-processor designs because of its superior resource utilization. Because the MINT package provides only a simulation of the instruction set of the MIPS R3000 processor, which is a RISC machine, we add a simulation of the PMP-MP system in SEESMA.

The most important feature of PMP-MP is that several parallel running threads share only one cache in each processing element. Consequently, to simulate the PMP-MP architecture, when the front end invokes a back-end function that will access the shared cache, the thread identifier[1] for each task must be mapped to the identifier of the processing element in which the thread resides. On the other hand, when a task is going to return a T_ADVANCE value to the front end, we have to remap the identifier to its original thread identifier to let the thread issuing the memory access continue execution.

## 2. Memory Consistency Models

We provide simulations of four conventional hardware-centric memory consistency models, including sequential consistency (Lamport, 1979), processor consistency (Goodman, 1989), weak consistency (Dubois *et al.*, 1986), and release consistency (Gharachorloo *et al.*, 1990). To simulate these four memory consistency models in SEESMA, we categorize and label memory accesses into acquire, release, ordinary read, and ordinary write classes. To label memory accesses in source benchmark programs, we insert the MINT-supplied generate_event() functions with different parameters into the LOCK, UNLOCK, and BARRIER macros, respectively. Calls to generate_event() within the object program will cause an event to be generated for the current running thread. The back-end sim_user() is called, and the arguments for generate_event() can be retrieved from the event structure to label the type of individual synchronization access. Although we label synchronization accesses in four classes, at the time of simulation, we have to map these four classes into the original ones as illustrated in Table 1 according to the simulated memory model.

In addition to the four conventional models, we provide a simulation of our proposed memory consistency model, called the PSC model (Wu and Chen, 1998b). The new model is an extension of the release consistency model, particularly for PMP-MP systems. It uses a new method of categorization for memory accesses and utilizes the features of PMPs. We further partition acquire and release accesses into three subcategories: one for lock-unlock pairs, one for barrier synchronization, and the last for other accesses. According to the semantics of each synchronization primitive, each sub-category has its own relaxed restrictions. On the other hand, the feature of a PMP is that it is capable of executing more than one thread at the same time where all parallel threads share only one cache

[1] In fact, it is called the process identifier in the MINT package.

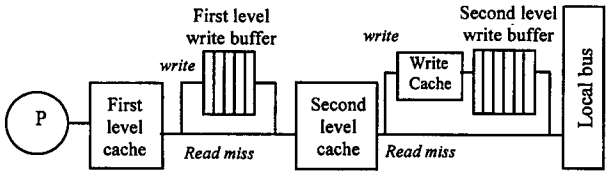


**Fig. 4.** The processor environment with a write cache.

hierarchy. Under the new model, we can use *dual write-caches* to reduce write traffic and synchronization time.

## 3. Two-Level Cache with Write Cache

The processor SEESMA simulates is blocked on read misses but not on write misses (Dahlgren and Stenstrom, 1995). That is, the back-end will return T_ADVANCE to the front-end only when write misses occur. Read accesses are handled as follows. When a read reference hits on the FLC, it returns the requested words to the processor. When a read miss occurs on the FLC, we have to check whether the requested block is buffered in the FLWB. If the answer is yes, the read miss request can not be sent to the SLC until all the write references buffered in the FLWB have been issued to the SLC. This handling procedure enables the FLC to simply and quickly support one-cycle accesses. If the requested block is not in the FLWB, the read miss is sent to the SLC. The requested words are returned either from the SLWB if they are found in the SLWB or from the SLC when the read request hits on the SLC. Otherwise, a global action occurs for the read request. A replacement prior to the global action may be necessary.

On the other hand, we handle write accesses as follows. A write request from the processor is blocked when the FLWB is full. Once the FLWB has a free entry, the write request is pushed to the FLWB. Meanwhile, the contents of the words on the FLC are updated if the write request hits on the FLC. Because the FLWB is a FIFO (First-In First-Out) queue, a write request can not be issued to the SLC until all the write accesses prior to it have been sent to the SLC. Whenever a write reference accesses the SLC, it will issue a global memory action (e.g., write invalidation request) which is buffered in the SLWB. The global action takes place when the local bus can transmit it.

The default access times to the FLC and SLC are set to 1 and 3 processor cycle times. However, the penalties of various access misses are not fixed to default times. They depend on many factors, e.g., the state of the cache block, the location of the memory copy, the routing time in the interconnection network etc.

In addition, we add a simulation of write cache (WC) (Dahlgren and Stenstrom, 1995) as an option. Figure 4 shows the processor environment after a write cache is added. Whenever a write reference accesses the SLC, it will issue a global memory action, and the action will be buffered in the WC instead of the SLWB. In addition, actions belonging to the same cache block will be merged into a single one in the WC. The actions in the WC will be forwarded to the SLWB because of WC block replacement.

Memory access ordering requirements must be enforced by the underlying memory consistency model. Sequential consistency and processor consistency require that before any one processor can perform a write access, all previous write accesses in the program order must first be performed. Consequently, write caches are not supported under these two models because there is no opportunity for the WC to merge write requests. However, under weak consistency models, every ordinary access can be performed independently of other ordinary accesses, so write caches must only be flushed upon arrival of a synchronization reference. Therefore, write caches can merge write access requests between any two adjacent synchronization references. Consequently, the amount of network traffic due to global memory actions is reduced. Similarly, in release consistent systems, write caches must only be flushed upon arrival of a release reference.

## 4. Cache Coherence Protocol and Directory Structure

We support three types of cache directory structures: fully-mapped centralized, limited centralized, and distributed directories (Stenstrom, 1990). Under each of these three directory structures, simulations of several types of cache coherence protocols are provided in SEESMA. As reported by Dahlgren and Stenstrom (1995), cache coherence protocols can be classified into four types: write-invalidate, write-update, clean, and competitive-update. Clean protocols are similar to write-invalidate protocols except that they update the memory copy if there are copies in caches other than that from which the write request came. Competitive-update protocols are similar to write-update protocols except that a cache block is invalidated when it has been updated a fixed number of times, called the competitive threshold, without intervening local accesses.

The centralized directory-based cache coherence protocols we simulate are similar to that described by Dahlgren and Stenstrom (1995) while the distributed directory-based protocols are extensions of the IEEE SCI standard (Su, 1996).

### 5. Interconnection Network

We provide simulations of local interconnection and global interconnection networks. The local interconnection provides the communication media between the two-level cache controller, the local shared memory controller, and the network controller. It can serve one request at a time. Therefore, it has to use an arbiter to select a request when there are multiple requests. The arbitration we simulate selects the first request in the task queue.

The global interconnection network is the interface between any two nodes as shown in Fig. 1. The topology we simulate is a $k$-ary $n$-cube interconnection network (Dally, 1990). A $k$-ary $n$-cube network has $n$ dimensions and totally $N$ nodes, where $N=k^n$. A node in the $k$-ary $n$-cube can be identified by an $n$-digit radix $k$ address, $a_0$, ..., $a_{n-1}$, where $a_i$ represents the node's position in the $i$th dimension. Each node can forward messages to its upper neighbor in each dimension. For example, the address of the upper neighbor in the $i$th dimension is $a_0$, ..., $a_i+1 \pmod k$, ..., $a_{n-1}$. "Most concurrent computers have been built using networks that are either $k$-ary $n$-cubes or are isomorphic to $k$-ary $n$-cubes: rings, meshes, tori, direct and indirect binary $n$-cubes, and Omega network" (Dally, 1990). For instance, a 16-ary 1-cube is isomorphic to a 16-node ring network. Therefore, we have chosen this type of interconnection network for simulation in SEESMA.

Each node has two packet buffers: an input packet buffer and an output packet buffer. A packet that the input packet buffer receives may have either arrived or not arrived at its destination node. In the former case, the packet is transferred to the local interconnection in the corresponding node. Otherwise, the packet is put into the output packet buffer.

The packets that the output packet buffer receives may be those that have been issued from other nodes but have not arrived at their destination nodes, or those that have been issued from the local node. For each packet in the output buffer, the router will calculate the next node to which the packet is to be transferred according to the routing algorithm. Then the packet contends with others for the use of the routing path to the next node. The arbitration algorithm is first-in first-out. Next, the packet transfer process will calculate the time needed for the transfer according to the packet size. After this time period, it will put this packet into the input packet buffer of the next node.

The parameters which we can adjust for the interconnection network in SEESMA are: (1) the radix, (2) dimension, (3) switch delay, (4) wire delay of a link, (5) link width, and (6) unidirectional or bidirectional transmission.

## IV. Graphic Interface Capabilities

SEESMA supports a friendly graphic user interface (GUI) which enables users to easily and efficiently operate the simulation and evaluation environment. Our GUI interface has two parts: an input and an output interface, which are described in the following subsections.

### 1. Input Interface

The architecture characteristics are regarded as parameters. Users are asked to input a list of command line options to specify the parameters of the simulation environment and the simulated architecture when a text-based user interface is used. Each option is prefixed by an identifier followed by an input value. We give below an example to show how a command line can be written to execute the MINT simulator.

```
../OMSS/cache -ts -r -p64 — -p64 -N16 -n2 -k4 -f16 -s32
-W8 -Pc -T4 -o/usr/home/ccwu/data/mp3d/p64N16_Comp4_
w8 mp3d 50000 64 test.geom
```

Obviously, this type of input suffers from the following disadvantages:

(1) The command line is so long that it is tedious and inconvenient for users to input the command line for each simulation.
(2) We have to remember the default value for each parameter to determine which options we must specify.
(3) It is easy to confuse or hard to remember the corresponding identifiers for various command line options.

To make it easy for users to run SEESMA, we have designed a friendly graphical input interface as shown in Fig. 5. The interface has three parts: a menu bar, an architecture graph, and a status region. The menu bar consists of five main selection items; each item has its own menu hierarchy according to the category of the simulation parameter. The architecture graph shows the corresponding architecture whenever the user sets the architectural parameters. For instance, if we incorporate write caches into the system, there will be a connection between the write cache and the cache hierarchy in the architecture overview; otherwise, no connection will exist. Another example is that we can change the architecture as presented in Fig. 5 into a PMP-MP with write caches as shown in Fig. 6. Through real time display of the architecture graph, we can easily determine what kind of architecture we have chosen for simulation. The status region displays the parameters of an architecture component at which the cursor positions. For example, the status region shows

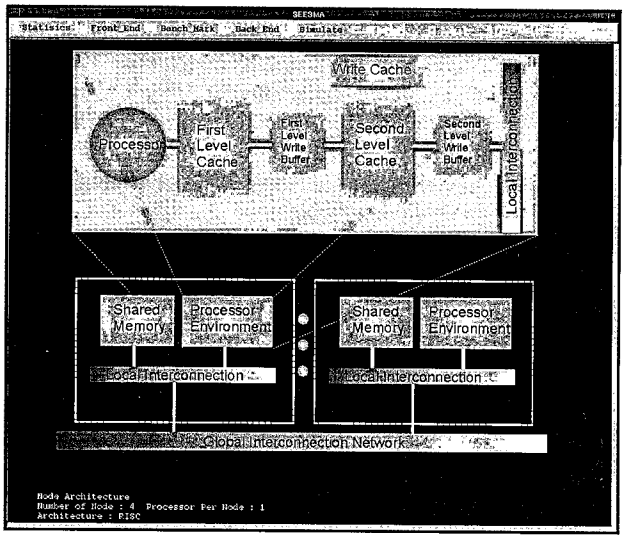


**Fig. 5.** Graphical user input interface.

the current parameter setup for the node architecture when the cursor moves to the area of the processor environment as shown in Fig. 6. By means of the status region, we can easily determine the entire parameter setup by moving the cursor. If we find an incorrect setup, we can just click on the position of the architecture block to change the setup.

SEESMA provides two input methods: menu based input and graphic based input. With menu based input, the selection items are collated as a pull down menu on the menu bar, and the dialogue window of each item is then presented as a pop up menu. With graphic based input, the user is directed to a dialogue window without looking through the pull down menu. For each input, the user can just click on the appropriate position of the architecture block. In Appendix 1, we present the complete procedure for an example to illustrate the input capabilities.

## 2. Output Interface

After each simulation, the traditional simulation environment outputs the simulation results in text form on the screen or into a file. Usually, to study an interesting topic, we will run several simulations by changing the setup of an architectural parameter, and then comparing the simulation results. However, from text-based simulation results, we can not fully understand the differences in the results. As a result, we produce a graph of statistics after collecting data from these simulation results to illustrate apparent differences. Therefore, we have designed an output interface to facilitate comparison of the results. We provide bar and curve graphs, and the details are described in the following section.

# V. Performance Evaluations of MP in SEESMA: An Illustration

In this section, we will give a complete design example to illustrate the procedures for using SEESMA to reveal some important issues and performance gains in designing specific multiprocessor architectures.

Write caches can merge several write requests that will access to the same block into a single request. By cutting the number of write requests, write caches can enhance the performance of cache coherence protocols. As reported by Dahlgren and Stenstrom (1995), after write caches are added, competitive-update protocols are superior to any other protocol for multiprocessors with non-parallel-multithreaded processing elements. We investigate their effect on PMP-MP systems in this section.

We assume that all memory accesses to code and private data always hit on the FLC, and that each requires a single processor clock cycle. We summarize several important architecture parameters in Table 1 and others are as follows: (1) the FLC, SLC, and WC are all direct-mapped; (2) the memory pages are distributed in a round-robin fashion; (3) transmission is unidirectional; (4) release consistency is used as the memory consistency model; and (5) the cache directory structure is fully-mapped, centralized directory-based.

We need to point out some parameters which differ from those in Dahlgren and Stenstrom (1995) and influence performance. Because the total number of threads was 64 (16×4) in our study, the acquire-stall

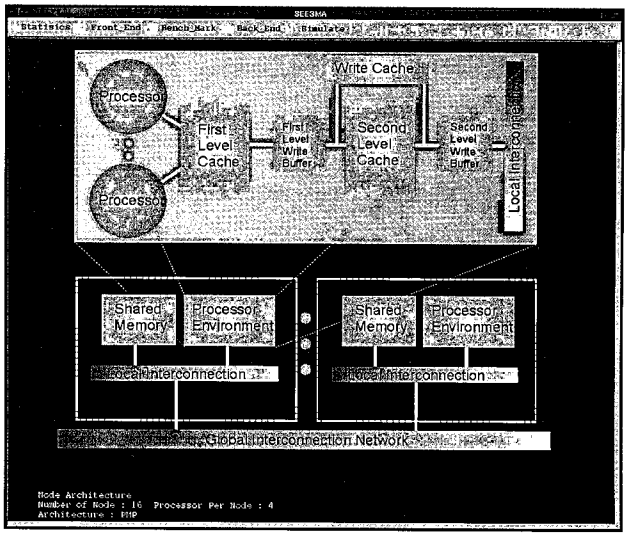


**Fig. 6.** Overview of the PMP-MP system with write caches.

**Table 2.** Architecture Parameters

| Parameter | Value | Parameter | Value |
|---|---|---|---|
| Number of Processing nodes | 16 | Number of threads per PE | 4 |
| Processor clock rate | 100 MHz | Size of FLC | 16 Kbytes |
| Size of SLC | 256 Kbytes | Access time to FLC | 1 pclock |
| Access time to SLC | 3 pclock[a] | Block size of FLC, SLC, WC | 32 bytes |
| Number of entries in FLWB | 16 | Number of entries in SLWB | 32 |
| Size of WC | 8 blocks | Size of memory page | 4 Kbytes |
| $k$-ary $n$-cube | $k$=4, $n$=2 | Linked width | 64 bits |
| Switch delay | 1 pclock | Wire delay | 1 pclock |

[a] processor cycle time

**Table 3.** Benchmark Programs

| Benchmark | Description | Data sets |
|---|---|---|
| MP3D | 3-D particle-based wind-tunnel simulator | 50K particles, 10 time steps |
| Cholesky | Cholesky factorization of a sparse matrix | The matrix bcsstk 14 |
| Pthor | Distributed time digital circuit simulator | RISC circuit, 1000 time step |
| Ocean | Ocean basin simulator | 130×130 grid, tolerance $10^{-7}$ |
| Barnes | Hierarchical N-body gravitation simulator | 1024 bodies, 3 steps |
| Water | N-body water molecular dynamics simulation | 343 molecules, 3 time steps |

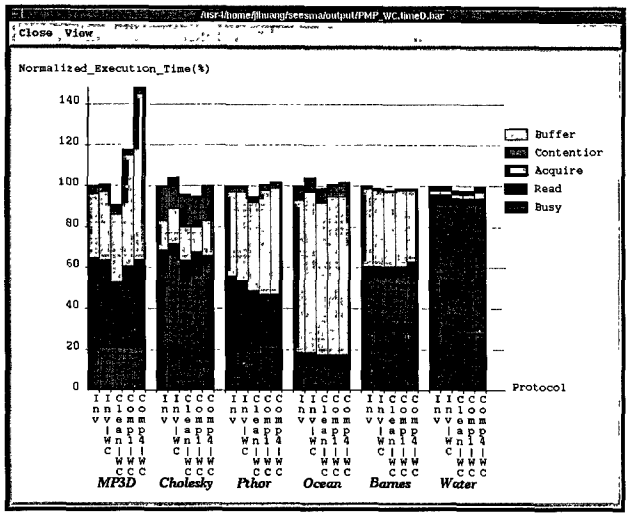


**Fig. 7.** Normalized execution times for various cache coherence protocols after augmentation with write caches.

time was perhaps much larger than that for the 16 threads in Dahlgren and Stenstrom (1995). In addition, because the SLC capacity was 256 Kbytes instead of infinite size in Dahlgren and Stenstrom (1995), we have to consider the impact of replacement on the SLC. A smaller cache increases the number of dirty blocks that must be replaced and the number of coherence misses accessing clean memory blocks under write-invalidate protocols, thus diminishing the advantage of the clean protocol.

We used six SPLASH applications (Singh *et al.*, 1992; Woo *et al.*, 1995) listed in Table 2 as benchmark programs. We briefly describe the behaviors of these benchmark programs in Appendix 2. For more details, we refer the reader to Singh *et al.* (1992) and Woo *et al.* (1995).

All applications were written in C using the ANL (Argonne National Laboratory) macros (Boyle *et al.*, 1987) and were compiled using cc under IRIS version 3 at optimization level 2 on an SGI workstation. Because the working set of these six programs are very small when compared to the size of the used FLC and SLC, the influence of replacements on performance is small. The effect of coherence maintenance is the primary factor that influences performance. All statistics presented in the following were gathered in the parallel sections of the benchmarks.

Figure 7 shows the performance improvement after adding write caches to write-invalidate (Inv-WC), clean (Clean-WC), and competitive-update with threshold T (CompT-WC) protocols. The execution time is the time interval from when the process 0 forks the first child process until all the child processes are joined. The execution time for each application and protocol was normalized relative to the execution time under Inv. In other words, the normalized execution time was derived by dividing the original execution time under a certain protocol by the original execution time under Inv. In addition, we divided each execution-time bar into five sections: busy time (bottom section);

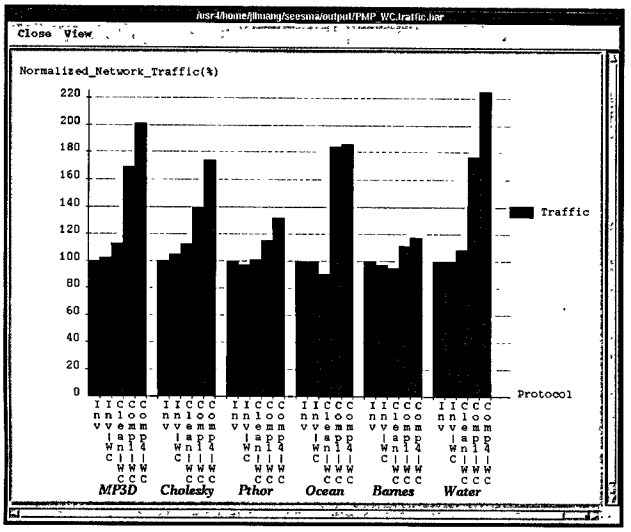


**Fig. 8.** Normalized network traffic for various cache coherence protocols after augmentation with write caches.

read-stall time (i.e., the time spent servicing cache misses); acquire-stall time (i.e., the time spent waiting for a lock to be acquired); contention time (i.e., the time spent waiting for access to the FLC); and buffer-stall time (i.e., the time the processor spent stalled due to a full FLWB). The most important observation is that Clean-WC outperformed Inv for all applications. The advantage of the Clean-WC protocol is the short read-penalties with a slight increase in network traffic. On the other hand, Inv-WC and Comp-WC were generally worse than Inv although the performance difference was small for three of the applications (Cholesky, Pthor, and Ocean). Inv-WC had longer acquire-stall times than Inv for the following reason.

Inv-WC delays the issuance of global writes in write caches until a synchronization point is encountered. The synchronization access then has to wait for the write cache to be flushed. In particular, release accesses cannot be issued until the write requests have all been flushed from the write cache and have all been performed. This waiting keeps other processes from entering the critical section and, thus, increases the acquire-stall time. On the other hand, the write-invalidate protocol without write caches sends only the first in a sequence of writes to the same block to the memory controller. Once a write-miss reply is returned (which means the cache controller has acquired ownership of that block), all other write misses are deleted after the block is updated according to the write ordering. The effect of this procedure is similar to the function of write caches: all write requests to the same block can be merged into a single one. As a result, adding write caches is not beneficial for the write-invalidate protocol.

For competitive-update protocols, each write request definitely incurs a global write. This feature has two extreme effects on write caches. First, write caches can considerably reduce network traffic by merging write requests; therefore, performance can be enhanced substantially. Second, there are usually many write requests to be flushed from the write cache whenever a synchronization access is encountered; thus, acquire-stall time is increased. We see from Fig. 7 that the second effect dominates the results. Competitive-update protocols have higher acquire-stall time though the network traffic is reduced drastically, as shown in Fig. 8. The network traffic is the total number of bits transferred in the interconnection network during the parallel section of each benchmark program. The normalized network traffic was derived by dividing the original network traffic under a certain protocol by the original network traffic under Inv.

Next, we will examine the different effects on total performance of various numbers of threads for each processing element. Figure 9 shows the performance speedup of each individual protocol relative to that of the write-invalidate protocol under a fixed number of threads per PE for MP3D. For the system with one thread per PE, the competitive-update protocol with write caches outperforms the write-invalidation protocol. However, when the number of threads per PE is increased, the competitive-update protocol with write caches may be worse than Inv. On the other hand, Clean-WC outperforms Inv for architectures with various numbers of threads per PE. Moreover, because the competitive-update protocol requires one counter per cache block, the clean protocol is more suitable for

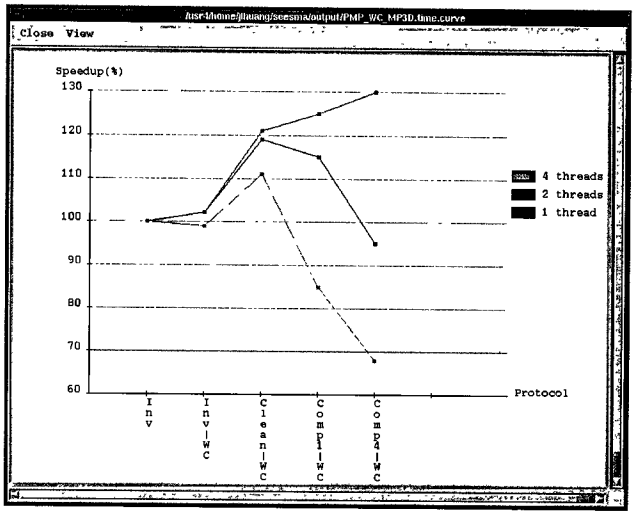


**Fig. 9.** Normalized execution times for various cache coherence protocols with write caches for various numbers of threads per PE.

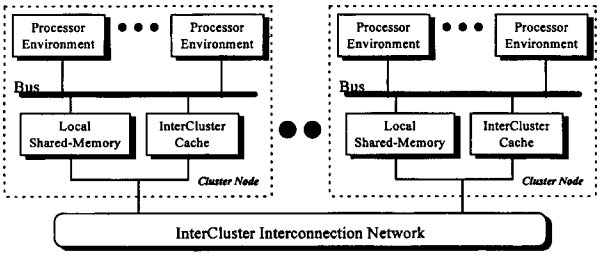


**Fig. 10.** Overview of clustering architecture.

PMP-MPs after the cost/performance tradeoff is considered.

In addition to the above example, we have used SEESMA to evaluate other design issues and to verify our proposed design methods as follows: (1) we have found that the clean cache coherence protocol usually has the best performance for multiprocessor systems no matter whether the processor element is multithreaded or not (Wu and Chen, 1998a); (2) the characteristics of write caches on PMP-MPs have been extensively evaluated (Wu and Chen, 1998a); (3) the performance of doubly linked directory cache coherence protocols in multiprocessor systems has been evaluated (Su, 1996); (4) the performance of our proposed optimization method for migratory-sharing accesses has been evaluated (Su *et al.*, 1996); (5) the performance of our proposed memory consistency model, the PSC model has been evaluated; and (6) the performance of our proposed parallelization technique has been evaluated (Pean and Chen, 1997).

## VI. Conclusions and Future Development

System architecture designers usually use a simulator not only to verify their proposals, but also to understand the interaction between system components. We have constructed a simulation environment called SEESMA, which is a platform for research and education on shared-memory multiprocessor architecture systems. It provides versatile simulations for architectural functions in an integrated environment with a friendly graphical user interface. The primary simulation functions include the following: (1) two CPU types: RISC and PMP architectures; (2) a two-level cache hierarchy with FLWB, SLWB, and write caches; (3) five cache coherence protocols: write invalidate, write update, competitive-update, and clean protocols; (4) three cache directory structures: centralized fully-mapped, centralized limited, and distributed SCI structures; (5) five memory consistency models: sequential, processor, weak, release, and our proposed PSC models; (6) K-ary and N-cube interconnection networks. To simulate a target system architecture, we can use the menu bar or just click on the architecture graph to set up the simulation and architectural parameters. The simulation results are dumped to a file. Optionally, we can set up the simulation parameters to view the specified evaluation results through graphs of statistics. In summary, SEESMA provides a good research environment for those who are interested in system designs for shared-memory multiprocessor architectures. We have also been constructing a web site at http://cspc4.csie.nctu.edu.tw that provides various information about SEESMA.

The multiprocessor architecture we have simulated so far has only one processor on each node. In the future, we will investigate the clustering architecture (Lenoski *et al.*, 1992) shown in Fig. 10. Several processor environments linked by a local bus are assembled as a cluster node, and then several cluster nodes linked by a intercluster interconnection network are brought together as a large system. Because there are several processors on a cluster node, we can design a node according to the characteristics and advantages of a small-scale multiprocessor system. On the other hand, the inter-cluster architecture design follows the design rationale for large-scale multiprocessor systems. The clustering architecture is attractive because it has the advantages of both small-scale and large-scale multiprocessor architectures.

The clustering architecture we will simulate in our future work will have the following features. Cache coherence among these processor environments maintained by a snoopy-based protocol. On the other hand, we will use the IEEE SCI standard to maintain cache coherence among cluster nodes. In addition, we will incorporate an inter-cluster cache on each cluster node to buffer the remote data in order to exploit the data locality among local processor environments. We will focus on the design of an interface between the snoopy-based protocol and the SCI protocol.

## Acknowledgment

The authors would like to thank the reviewers for their helpful comments. This research was supported by the National Science Council of the Republic of China under contract numbers: NSC 85-2221-E009-037 and NSC 86-2213-E009-094.

## Appendix 1
## An Illustrative Example of Running the SEESMA

In this appendix, we will show how the graphical interface can be used to input the parameters for simulation of the architecture with the clean protocol as described in Section V. Moreover, we will show how the parameters of the statistics graph can be specified

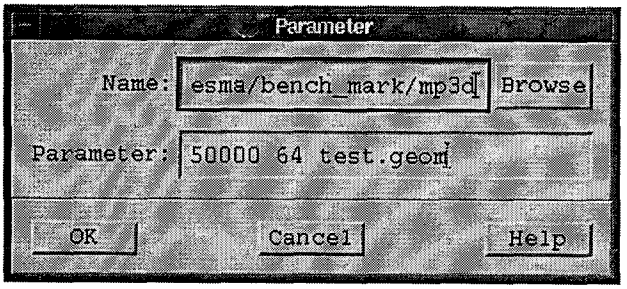


**Fig. 11.** Input the name and parameters of the benchmark program.

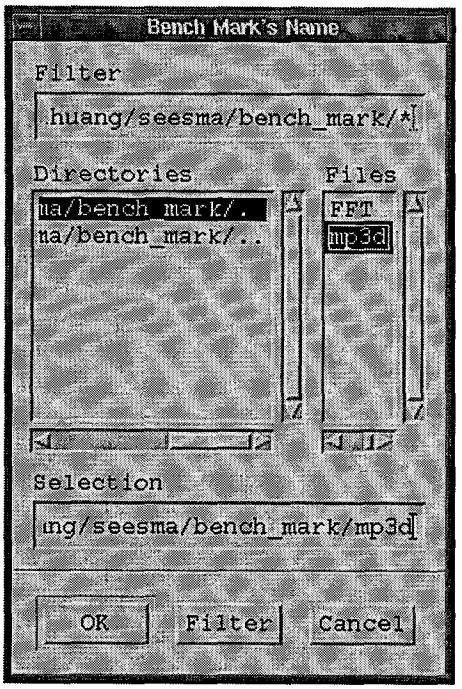


**Fig. 12.** Input the name of the benchmark program using the browser.

to construct a bar graph as shown in Fig. 7.

Step 1: Select the "Bench_Mark" button in the menu bar as shown in Fig. 5.

Step 2: Input the name and parameters of a benchmark program in the window as shown in Fig. 11 directory or click on the "Browse" button to select a benchmark program via the window as shown in Fig. 12. In this example, we choose MP3D as our benchmark program.

Step 3: Click on the "processor" graph as shown in Fig. 5 and input the parameters of the node architecture in the window as shown in Fig. 13.

Step 4: Click on the "First Level Cache" graph as shown in Fig. 5 and input the parameters of the two-level cache and two-level write-buffer in the window as shown in Fig. 14.

Step 5: Click on the left "Shared Memory" graph as shown in Fig. 5 and input the parameters of the cache coherence protocol and directory structure in the window as shown in Fig. 15. In this example, we choose the clean protocol under a centralized directory structure.

Step 6: Click on the right "Shared Memory" graph as shown in Fig. 5 and input the parameters of the memory consistency model in the window as shown in Fig. 16. In this example, we choose the release consistency model.

Step 7: Click on the "Write Cache" graph as shown in Fig. 5 and input the size of the write cache in the window as shown in Fig. 17.

Step 8: Select the "Back_End" button in the menu bar as shown in Fig. 5.

Step 9: Select the "Output" item in the pull-down menu as shown in Fig. 18.

Step 10: Input the parameters of the statistics graph in the window as shown in Fig. 19. The parameters are set according to the kind of statistics graph we desire to obtain after several simulations. In this example, we wish to produce the graph shown in Fig. 7. Therefore, the parameters have to be consistent with the simulation parameters as specified above. In this example, the benchmark program is MP3D, the cache coherence is maintained by clean protocol, and the system has write caches.

## Appendix 2
## Brief Descriptions of Six Benchmark Programs

In this appendix, we will briefly describe the behaviors of the six benchmark programs discussed in Section V.

MP3D (Singh *et al.*, 1992) is a 3-dimensional particle-based wind tunnel simulator. There are two types of data structures. (1) The particle array holds the molecules and records their positions and their velocities. (2) The space array represents the physical space, the boundary conditions, and the flying objects. It evaluates the

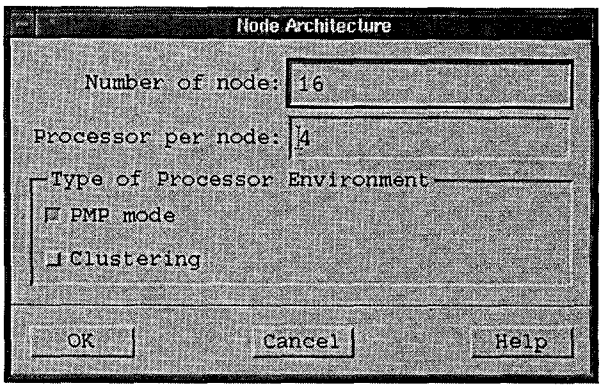


**Fig. 13.** Input the parameters of the node architecture.

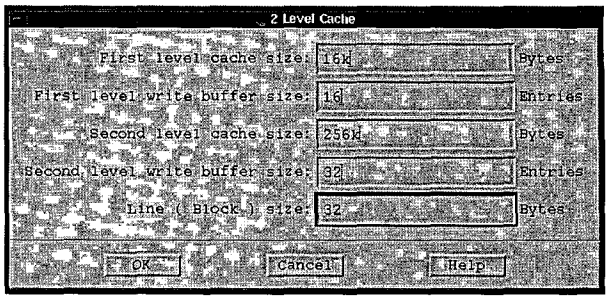


**Fig. 14.** Input the parameters of the two-level cache and write buffers.

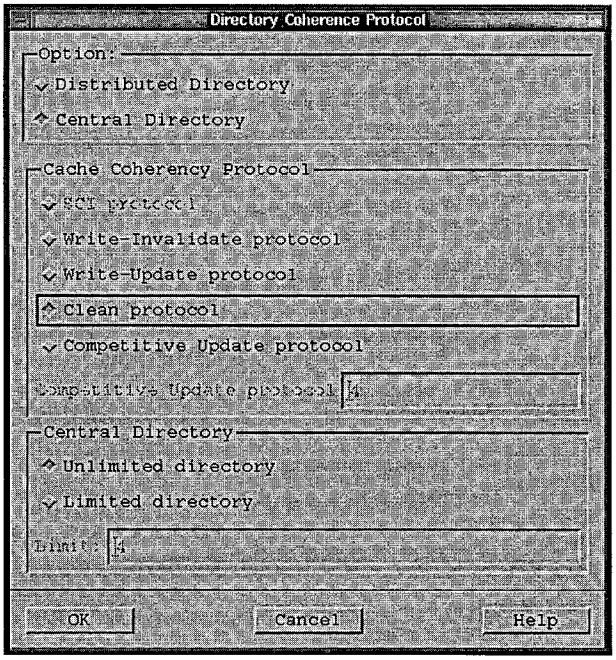


**Fig. 15.** Input the cache coherence protocol and the directory structure.

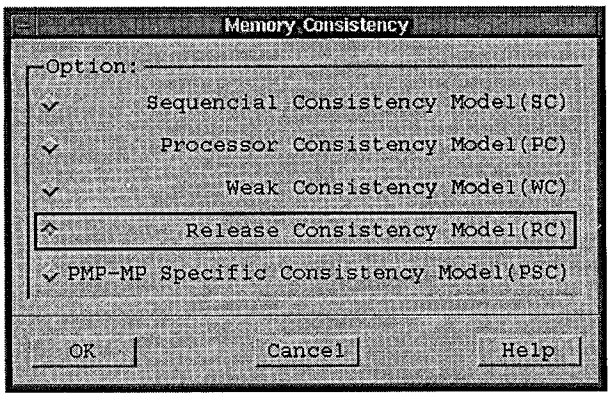


**Fig. 16.** Select the memory consistency model.

positions and velocities of molecules over a sequence of time steps. During each time step, molecules are picked up one at a time and moved according to their velocity vectors. The program is parallelized by statically dividing the particles equally among the processes. Barrier synchronization is inserted between each time step.

Cholesky (Singh *et al.*, 1992) performs parallel Cholesky factorization of a sparse matrix. The matrix is divided into supernodes, which are sets of columns with identical non-zero structures. The supernodes are further divided into conveniently sized chunks of columns, called panels. A panel receives updates from other panels to its left in the matrix. After receiving all the updates, it is placed in the task queue. A free process removes the panel from the task queue and performs the associated modification. This in turn yields other panels to be placed on the task queue. The main data structure is the sparse matrix, and the principle operation is the repeated addition of a multiple of one column to another column. Contention occurs for the task queue and for column modification, which are protected by locks.

Pthor (Singh *et al.*, 1992) is a parallel distributed-time logic simulator. The main data structures are the logic elements, the nets, and the task queues. Each element has a preferred task queue to increase data locality. Each process performs the following loop. It removes an active element from its task queue and determines the changes on that element's outputs. It then looks up the net data structure to schedule the newly activated elements in the task queues.

Ocean (Woo *et al.*, 1995) simulates the role of eddy and boundary currents in influencing large-scale ocean movements. The main data structures are four-dimensional (4-D) arrays, which are comprised of grids. The program partitions the grids into square-like subgrids. Each processor is assigned a set of contiguous and local subgrids. In each time step, each processor solves a set of spatial partial differential equations using a red-black Guass-Seidel multigrid equation solver. The simulation is performed for many time-steps until the eddies and mean ocean flow attain a mutual balance.

Barnes (Singh *et al.*, 1992) performs a gravitational N-body simulation. The primary data structure is an 8-ary tree, which is implemented as an array of bodies and an array of space cells that are linked together. Bodies are statically assigned to processors for the duration of a time-step. During each time step, each processor calculates the forces exerted on its own subset of bodies. The bodies

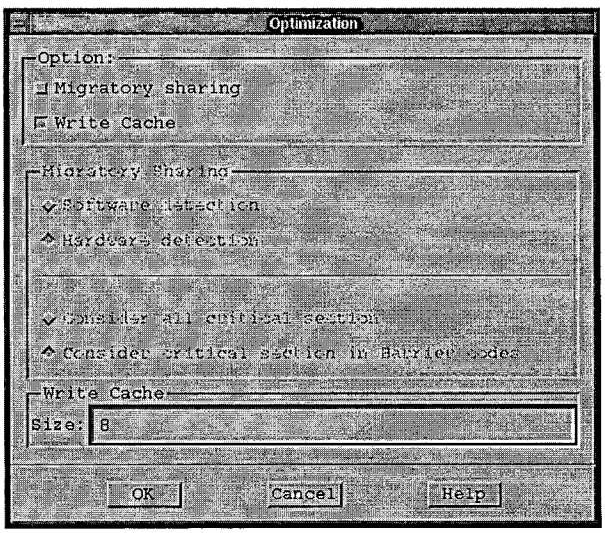


**Fig. 17.** Input the parameter of the write cache.

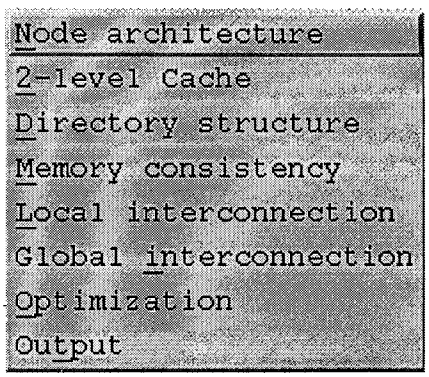


**Fig. 18.** Menu of the "Back_End" item.

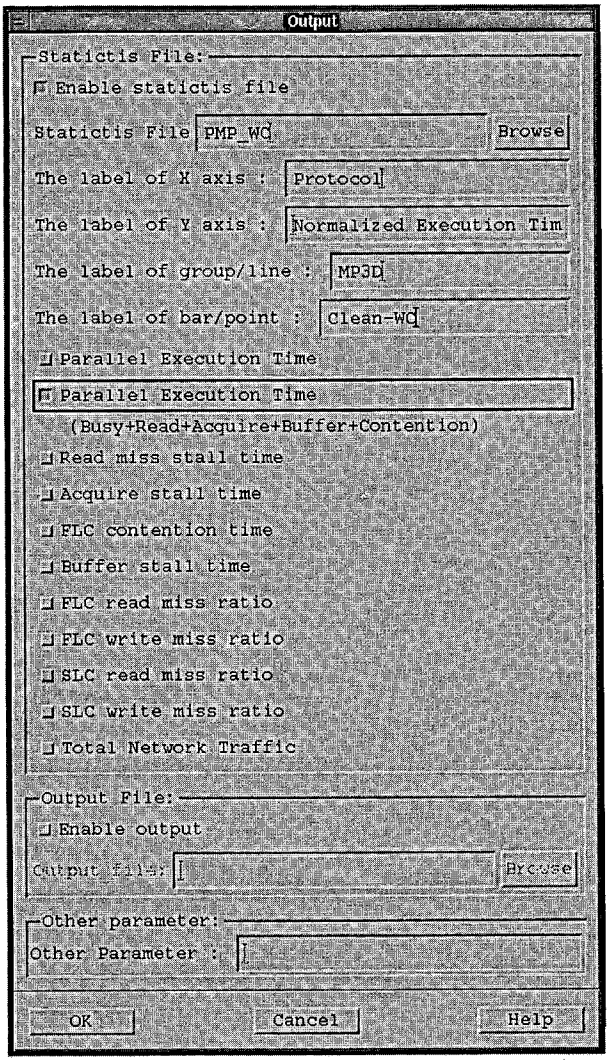


**Fig. 19.** Input the parameters of the statistics graphs.

are then moved under the influence of those forces. Finally, the tree is regenerated for the next time step. A set of distributed locks is used for exclusive access. In addition, barriers are used to separate different computation phases and successive time steps.

Water (Singh *et al.*, 1992) simulates the evolution of a system of water molecules. The principal data structure is a large array of records that store the state of each molecule. The molecules are statically split among processors. During each time step, each processor calculates the interaction of the molecules with one another. For each molecule, each processor calculates the interactions between a molecule it owns and half of the other molecules. Synchronization is maintained by means of locks on the molecules being modified, and through barriers that separate different computation phases.

## References


Boyle, J., R. Bulter, T. Disz, B. Glickfeld, E. Luck, R. Overbeek, J. Patterson, and R. Stevens (1987) *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, Inc., New York, NY, U.S.A.

Dahlgren , F. and P. Stenstrom (1995) Using write caches to improve performance of cache coherence protocols in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, **2**(26), 193-210.

Dally, W. J. (1990) Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, **39**(6), 775-785.

Davis, H., S. Goldschmidt, and J. Hennessy (1991) Multiprocessor simulation and tracing using Tango. *Proceedings of the International Conference on Parallel Processing*, pp. 99-107. St. Charles, IL, U.S.A.

Dubois, M., C. Scheurich, and F. Briggs (1986) Memory access buffering in multiprocessors. *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 434-442. Tokyo, Japan.

Eggers, S. J., D. R. Keppel, E. J. Koldinger, and H. M. Levy (1990) Techniques for efficient inline tracing on a shared-memory multiprocessor. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 34-47. Boulder, CO, U.S.A.

Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy (1990) Memory consistency and event ordering in scalable shared-memory multiprocessors. *Proceedings of the 17th Annual International Symposium on Computer Architectures*, pp.15-26. Seattle, WA, U.S.A.

Gjessing, S., D. B. Gustavson, J. R. Goodman, D. V. James, and E. H. Kristiansen (1991) The SCI cache coherence protocol. In: *Scalable Shared Memory Multiprocessor.* M. Dubois and S. Thakkar Eds. Kluwer Academic Publishers, Norwell, MA, U.S.A.

Goodman, J. R. (1989) *Cache Consistency and Sequential Consistency.* Computer Sciences Technical Report #1006, Computer Sciences Department, University of Wisconsin, Madison, WI, U.S.A.

Grahn, H., P. Stenstrom, and M. Dubois (1995) Implementation and evaluation of update-based cache protocols under relaxed memory consistency models. *Future Generation Computer Systems*, **11**(3), 247-271.

Hirata, H., K. Kimura, S. Nagamine, and Y. Mochizuki (1992) An elementary processor architecture with simultaneous instruction issuing from multiple threads, *Proceedings of the 19th International Symposium on Computer Architecture*, pp.136-145. Gold Coast, Qld., Australia.

Iannucci, R. A., G. R. Gao, R. H. Halstead, and B. Smith (1993) *Multithreading: A Summary of the State of the Art.* Kluwer Academic Publishers, Norwell, MA, U.S.A.

Lamport, L. (1979) How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, **C-28**(9), 241-248.

Lenoski, D., J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy (1992) The DASH prototype: implementation and performance. *Proceedings of the 19th International Symposium on Computer Architecture*, pp.92-103. Gold Coast, Qld., Australia.

Pean, D. L. and C. Chen (1997) An effective parallelization technique for nonuniform loops. *Proceedings of National Computer Symposium*, Vol. 3, pp. E58-E66. Taiwan, R.O.C.

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

Stenstrom, P. (1990) A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, **23**(6), 12-24.

Su, J. P. (1996) *A Study of Memory Subsystem Design for Multiprocessor System and Implementation of Its Simulation and Evaluation Environment.* Master Thesis. National Chiao Tung University, Hsinchu, Taiwan, R.O.C.

Su, J. P., C. C. Wu, and C. Chen (1996) Reducing the overhead of migratory-sharing access for the linked-based directory coherence protocols in shared-memory multiprocessor systems. *Proceedings of International Computer Symposium*, pp. 160-167. Taiwan, R.O.C.

Veenstra, J. E. and R. J. Fowler (1994) *MINT Tutorial and User Manual*. Technical Report No. 452, The University of Rochester, New York, NY, U.S.A.

Woo, S. C., M. Ohara, E. Torrie, J. P. Singh, and A. Gupta (1995) The SPLASH-2 programs: characterization and methodological considerations. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 24-36. Santa Margherita Ligure, Italy.

Wu, C. C. and C. Chen (1998a) A performance study on cache coherence protocols and write caches for parallel-multithreaded shared-memory multiprocessors. *Journal of the Chinese Institute of Engineers,* **21**(1), 33-46.

Wu, C. C. and C. Chen (1998b) A new relaxed memory consistency model for shared-memory multiprocessors with parallel-multithreaded processing elements. *Journal of Information Science and Engineering* (accepted).

# SEESMA:一個專供研究共享記憶體多處理機架構之模擬評估環境

伍朝欽　平德林　蘇仁彬　巫嘉榮　黃宣偉　黃俊龍　李佳霖　蔡慧婷　陳正

國立交通大學資訊工程研究所

## 摘　要

我們已建立了一個名稱是SEESMA的模擬評估環境。它可提供對共享記憶體多處理機架構有興趣的研究者一個探討及設計的工作平台。SEESMA是採用程式驅動的模擬器，並且由記憶體存取產生器及目標系統模擬器兩大部份所組成。前者基本上是MINT軟體程式，主要的功能是模擬應用程式在多個處理機中如何執行；而後者則主要是模擬記憶體子系統與連接網路的行爲。當前者執行到一個記憶體存取，它會呼叫後者以模擬此存取在記憶體子系統的行爲。一旦模擬結束，後者立刻通知前者可以繼續執行下去。

目前，SEESMA主要的模擬功能包括：兩種處理機架構、五種記憶體一致性模式、五種快取記憶體一致性協定、三種目錄架構、兩階層快取記憶體、以及連接網路等等。另一方面，我們提供圖形介面與線上的輔助系統以方便使用者操作此一模擬系統。在輸入方面，我們可利用滑鼠在架構圖上移動以了解各個功能單元的參數設定值，並可直接在圖上直接修改其相對的參數設定。隨著參數的變更，架構圖也會做相對的改變。至於輸出方面，SEESMA可將多次的執行結果以直條圖或曲線圖畫出，如此方便比較彼此間的差異性。目前，我們已可利用SEESMA從事於多處理機系統許多重要設計議題的探討及評估工作。它確實提供了此方面教學與研究或系統工程設計過程中一個良好的軟體平台。