Environment for Multiprocessor Simulator Development

Masaki Wakabayashi† Hideharu Amano†

†Department of Computer Science, Keio University
3-14-1, Hiyoshi, Kohoku-ku, Yokohama, Kanagawa Pref. 223-8522 Japan.
{*masaki,hunga*}*@am.ics.keio.ac.jp*

Abstract

Performance estimation is essential for designing and investigating of new architectures including multiprocessors. Software simulation is one of the most common methods, since there is no limitation on device technology nor hardware configuration. Although lots of software simulators have been developed and used, they must be modified for each distinct target system. For designers of new architectures, it is often a cumbersome job.

ISIS, an architecture independent simulation kit for multiprocessors, is developed so as to reduce such designers load. It includes various small simulators called "Units" corresponding to processors, buses, memories, caches and I/O devices. ISIS users can build simulators for their original target architectures only by connecting "Units" each other. The implementation cost is much reduced with little runtime overhead.

A sample instruction-level multiprocessor simulator which has 4 processors can be executed 40,000 to 70,000 steps per second. This paper also reports experimental results of ISIS in various research projects.

1. Introduction

In order to keep up drastic performance improvement of recent computer systems, parallel machines have been widely investigated and developed. For development of new computers, performance estimation before implementation is a key to success. Software simulation is one of the most effective methods owing to the flexibility and extendibility.

Software simulation is roughly classified into four types: probabilistic-model simulation, trace-driven simulation, execution-driven simulation and instruction-level simulation. For performance estimation, one of these methods is chosen depending on target machine architecture and required accuracy, and then simulation system is started to build. Various kind of simulators, e.g. ATUM[9], Tango[1], MINT[7] and SimOS[8] have been implemented and some of them are widely utilized in various research projects.

However, most existing systems are implemented for performance evaluation of a specific type of target multiprocessors. Therefore if the target multiprocessor is not a similar type of existing simulators' targets, researchers must modify the existing simulator or in some cases, an original simulator must be developed from scratch. This overhead of implementation is sometimes cumbersome job for researchers. Nevertheless, traditional simulators tend to reduce the runtime cost rather than implementation cost.

Here, we propose an architecture independent simulation kit for multiprocessors, called ISIS. Under different intention from traditional simulators, it aims mainly to reduce the implementation cost for building various simulators for various types of multiprocessors with various simulation schemes. In ISIS, there are many parts called *Units* each of which simulates a function block in multiprocessors and constitutes a library. A specific simulator for a multiprocessor can be formed only by connecting *Units* in a library. As well as reduction of the implementation cost, ISIS is designed so as not to degrade the simulation speed compared with traditional simulators. ISIS has been used in several research and development projects of multiprocessors, and *Units* in ISIS has been extended and brushed up.

Section 2 and Section 3 describe the design and the implementation of ISIS. Section 4 assesses the performance of ISIS and some simulator examples implemented with it. The precedent of research with ISIS is covered in Section 5, and Section 6 offers concluding remarks.

2. A Simulation Kit ISIS

2.1. Design Concept

Various simulation schemes are applied to multiprocessor simulators depending on both the target machine situation (architecture and scale) and simulation requirements (accuracy and simulation speed). It is a principal trade-off in simulation requirements, that is, accuracy versus simulation speed. Followings are common simulation schemes ordered in high execution speed but low accuracy: probabilistic-model simulation, trace-driven simulation[9], execution-

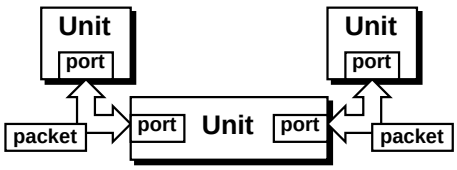

Figure 1. System model.

driven simulation[7], and instruction-level simulation[10]. These schemes are sometimes dynamically changed in order to fit the requirement[8]. Since ISIS is designed so as to reduce the simulator development cost for various target parallel machines, various simulation schemes must be supported as possible.

On the other hand, simulators can be also classified by its synchronization schemes: event-driven synchronization and clock-driven synchronization. The former is favorable when there are less event occurrences, while the latter is advantageous for target machines consisting of a lot of frequent working elements. Unlike the simulation schemes, we select only the clock-driven synchronization in ISIS. Using a common clock, a large simulator can be built with a lot of parts each of which works a small simulator itself just by connecting them.

Based on these considerations, we propose a simulation development kit called ISIS, which consists of many simulators of small functional blocks in multiprocessors. Each small functional block simulator called *Unit* works in clock-driven synchronization and can be easily connected through a standard interface. Using this strategy, a simulator for any target architectures can be developed easily to meet various simulation requirements. In ISIS, probabilistic-model simulation, trace-driven simulation, instruction-level simulation and their combination can be used. The execution-driven simulation scheme, which restricts the target is not included. It can be also combined in a specific type of library in future.

2.2. System Model

As shown in Figure 1, a simulator developed by ISIS consists of the functional block, the connection between blocks and the information sent/received. They are abstracted as *Unit*, *Port* and *Packet* respectively.

Unit: *Unit* is defined as a simulator of a function block such as processor, memory module, router and I/O device. It is implemented as an individual simulator, and works communicating with other *Units* through *Port* synchronized by a unique clock. For resolving the dependencies between events, the clock is divided into two phase: input and output. At the input phase, the information is transferred from outside to inside all *Units*, while it is sent from the inside in the output phase.

Port: *Port* is defined as the connection point of the communication path between *Units*. It manages all control processes for communication, and conceals them from internal implementation of *Units*. In order to avoid extra overhead, a quick communication method is used.

Packet: *Packet* represents an information transferred between *Units*. As all managements of the sent/received information are done by *Packet* itself, internal implementation of the *Port* can be done without considering actual data processing.

2.3. Implementation Cost

In order to reduce the implementation cost, *Unit* should be shared in various simulators as possible. We provide a lot of ready-to-use *Units* as a library. If *Unit* which fits to the purpose directly can not be found in the library, users must implement by themselves. Even in this case, most of required *Units* can be implemented with some modification of *Units* in the library. Using object oriented language, ISIS makes it easy to make the derived class from the basis *Units* in the library.

2.4. Runtime Cost

A runtime cost of a simulator is depending on the performances of each *Unit* and additional overheads for combining *Units*. In ISIS, the latter overhead is reduced as possible by tuning up the code of *Packet* and *Port*.

Furthermore another important issue which reduce the runtime cost in some cases is portability of simulator. When a target is simulated under various parameters, a lot of computers can work in parallel. For such a parameter survey, ISIS is designed to work in various platforms as possible.

3. Implementation

In this section, implementation of the system model and basic components: *Unit*, *Packet* and *Port* are described. We also introduce outlines of actual function block simulators implemented with the basic components in ISIS.

3.1. Description Language

In order to satisfy both the programming paradigm for large-scale design and execution speed, ANSI C++ is selected as an implementation language of ISIS. *Units*, *Ports* and *Packets* described in Section 2.2 are defined as classes, and they are supported to users as parts of the class library.

In order to reduce the implementation cost of each class, common characteristics between some classes are abstracted as base classes, thus they form class hierarchies. *Units*, *Ports* and *Packets* are the most fundamental constituent of

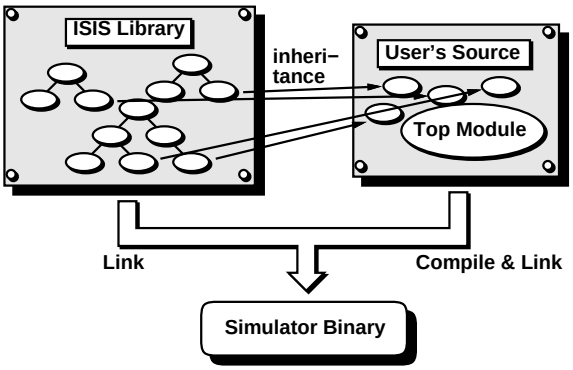

Figure 2. Structure of ISIS library.

```
class packet {
public:
  virtual packet* new_packet() const = 0;
};
```

Figure 3. Definition of packet class.

simulators and each of them integrates an individual class hierarchy. A simulator designer picks up desired class from the library, and uses it directly or makes a derived class if needed. Using inheritance facility, it is easy to make a derived class in most cases.

Figure 2 shows the structure of the library. The designed simulator is transformed to executable binary program through compiling and linking.

3.2. Implementation of Basic Components

Here, implementation of three basic classes *Unit*, *Port* and *Packet* are described. Another class *Device* is also introduce to reduce the implementation cost of *Unit*.

3.2.1. Packet

`packet` class is defined as a base class of *Packet* and the definition is shown in Figure 3. It is an abstract class, and has only a pure virtual member function `new_packet` to duplicate itself. Signal sets of buses or each packet passed through routers are defined as derived class of `packet`. High degree of flexibility is obtained for derived classes since it does not define any substance of physical packets.

3.2.2. Port

`port` class is defined as the base class of *Port* and the definition is shown in Figure 4. For example, I/O port of buses or links between router chips are defined as derived classes

```
class port {
public:
  void put(packet*);
  packet* get();
  void connect(port&);
  void disconnect();
  bool have_packet() const;
};
```

Figure 4. Definition of port class.

```
class unit {
public:
  virtual void clock_in() = 0;
  virtual void clock_out() = 0;
  virtual void reset() = 0;
};
```

Figure 5. Definition of unit class.

of `port`. It has some member functions — such as to connect/disconnect to each other (`connect`, `disconnect`), send/receive packets (`put`, `get`) and communication control (`have_packet`: checking packet existing). A virtual connection path, which can store only one packet, is cleated automatically when some ports are connected with each other. In order to improve the execution speed, packets are not copied in sending/receiving mechanism, but pointers for packets are copied.

Although parallel simulation is not possible to the current ISIS, the parallelization of the simulator can be easily extended if the internal communication control of `port` is modified for parallel execution.

3.2.3. Unit

`unit` class is defined as the base class of *Unit* and the definition is shown in Figure 5. It is an abstract class and has only three pure virtual member functions — state transitions at input/output phase (`clock_in`, `clock_out`) and state initialization (`reset`). Function blocks such as processors or memory modules are defined as derived classes of `unit`.

3.2.4. Device

Device is a class representing a small and simple element to support an implementation of a large scale *Units* like complicated processors.

The functional blocks without state transition are represented with *Devices*. Buffers used in cache, register files and instruction buffers are also classified into this class. By using defined *Devices*, a large scale *Unit* can be implemented with comparatively less cost. Now, almost 20 *Devices*

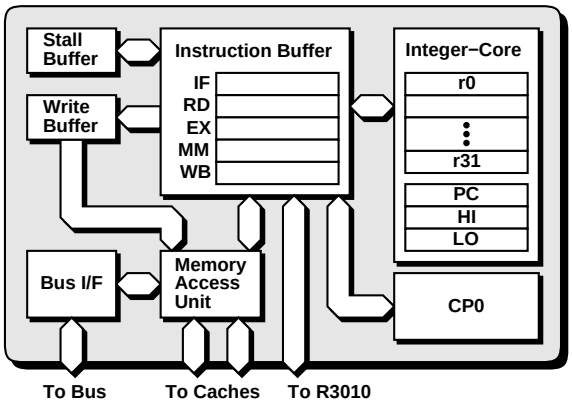

Figure 6. Structure of R3000 simulator.

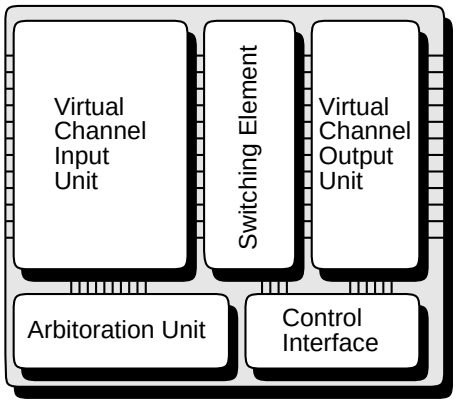

Figure 7. Structure of router.

are defined in the library. Unlike other classes in ISIS, class hierarchy is not formed for *Devices*.

3.3. Function Block Simulators

In the library, it is necessary to provide a large assortment of "useful parts" as well as the system model for reducing the implementation cost. Examples of "parts" which have been already implemented are introduced as followings.

3.3.1. Processor

MIPS R3000 processor simulator shown in Figure 6 is implemented as an example of existence processors.

It is almost complete clock-level simulator, and can simulate five-stage instruction pipeline, register file, primary caches, write buffer and bus interface exactly in clock-level. The clock-level simulator of a floating point operation coprocessor R3010 is also implemented.

3.3.2. Bus

Bus is a shared connection path between *Ports*, and can connect a lot of *Units* including processors or memory modules. We use derived classes from `port` class and `packet` class for the bus implementation. Bus packet class includes the information of an address line, a data line and a control line of an actual target bus. Bus port class has functions of sending/receiving packets and ownership control.

3.3.3. Cache

Since structure of the cache is highly depending on the processor connecting to it, ISIS does not support any specific cache units as its internal structure. Instead, ISIS supports many useful "parts of cache" as *Devices*: the model for a tag memory, data memory, bus interfaces and others.

3.3.4. Memory

The memory module simulator consists of two parts: the buffer part which keeps the whole data and its controller.

The buffer part manages page-based allocation mechanism. It does not allocate total memory size required in the simulator's code at the beginning of simulation, but allocates actually accessed by executed application dynamically. Thus no redundant resource is spent for the simulation even if the target machine requires more size of memory.

The controller part manages the data depending on the requirement of the bus. The detail of read/write access delay can be set here. The bus transaction including burst transfer and split transaction are also provided.

3.3.5. Router

This unit is defined as a clock-level simulator of a router which is an essential element of an interconnection network for multiprocessors. Figure 7 shows the structure of the router simulator.

All of following elements can be modified to support various target machines: input/output bandwidth, latency, number of virtual channel and buffer size. Both wormhole and virtual-cut-through routing are supported. In the router, virtual channel buffer, a crossbar and its arbiter are provided, and managed by the controller. Two important functions are not defined in base router class for leaving the simulator designers' choice: a routing algorithm function and an inner control function. As there are many sample functions of both algorithms in ISIS, usually designers are only required to select desired functions from them. Procedural description is also available.

3.3.6. Network Interface

This unit is defined as a clock-level simulator of a network interface. Bus used in the node processor and communica-

Table 1. Supported architecture and OS.

Architecture	OS
HP PA-RISC	HP-UX 10.10
Sun Sparc	SunOS 4.1.4 Solaris7
PC/AT compatibles	FreeBSD 3.5.1 Linux 2.2.17 Solaris8
SGI Origin	IRIX 6.4

tion between router are managed in it.

Since it is designed to support a distributed shared memory, ISIS can support an instruction-level simulation of NUMA.

3.3.7. I/O Device

I/O unit is defined for emulation of I/O functions in applications: file input/output, time management and so on. An instruction-level simulator developed with ISIS can execute a program including I/O operations with it. When an application issues a request packet to this unit, it issues the request to the operating system on the host machine.

3.4. Platforms and Applications

As mentioned in Section 3.1, we select ANSI C++ as an description language of ISIS. To support as many platforms as possible, no library is used except C++ standard libraries. Table 1 shows both the architectures and operating systems supported by current ISIS. Furthermore it works well with any other platforms if it has C++ compiler.

ISIS completely supports ANSI C, ANSI C++, ANSI FORTRAN 77 and FORTRAN 90 for applications on instruction-level simulators. GNU `gcc` and `newlib` are used for supporting environment of applications.

4. Performance

In this section, we present results of followings: required runtime memory, execution speed of a small simulator and the total performance of both instruction-level multiprocessor simulators and probabilistic-model one. Breakdown of source code size are also shown.

Here, simulation is executed on a PC/AT compatibles equipped with one 600MHz Pentium-III and 512MBytes main memory. Solaris8 is used as an operating system.

4.1. Required Memory

Table 2. Required memory size.

Unit	Memory (bytes)
R3000 processor	1,052
R3010 coprocessor	1,936
Memory controller	140
I/O unit and etc.	464
Router	244
Network I/F	1,052
Uniprocessor	3,668

Table 3. Speed of the uniprocessor simulator.

	loop	*copy*
Total steps	2,002,590	2,002,641
Total exec. time[sec]	6.34	5.37
Exec. time per step[μsec]	3.166	2.681
Number of steps per second	315,866	372,931

Table 2 shows runtime memory size in bytes required by elementary units. The “uniprocessor” at the bottom of Table 2 shows the simulator for an almost minimal scale computer consisting of a R3000 processor, a R3010 coprocessor, a memory module and an I/O unit without a router nor a network interface.

Table 2 shows that each unit requires small memory size and it means that ISIS has highly scalability at this point. Note that extra memory blocks are required to simulate caches or memory modules used in a target machine. However, as mentioned in Section 3.3.4, only the memory blocks are actually spent that are required by an executed application on simulators. A small extra memory for created *Packets* is also required at runtime.

4.2. Speed of the Minimal Simulator

We measured total simulation time of the instruction-level simulator for the uniprocessor described in Section 4.1. For reality, the target machine provides additional 16Kbytes primary instruction cache and 4Kbytes primary data cache.

Two simple applications are executed on the target: *loop* executes an empty loop with 1,000,000 iterations. *copy* executes memory-to-memory copy of 1Mbytes data. The results are shown in Table 3. Note that a simulation step is corresponding to the processor clock cycle.

Since frequent memory access accompanies a lot of processor wait, *copy* is executed faster than *loop*. This result shows that an instruction-level simulator can execute 310,000 to 380,000 steps of the target uniprocessor per second. As a clock-level simulator, this speed is enough useful.

Table 4. Workloads for instruction-level simulation.

Program	Problem size
BARNES	512 bodies
FFT	65,536 complex doubles
LU	256×256 matrix
OCEAN	66×66 grid
RADIX	2,097,152 keys

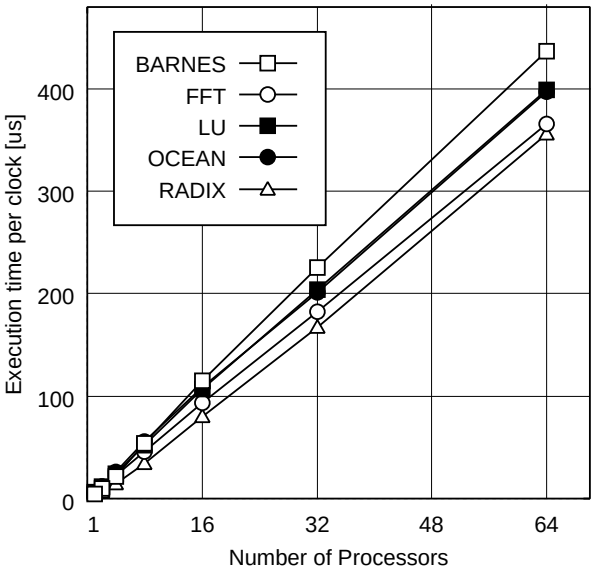

Figure 8. Performance of the instruction-level simulator.

Table 5. Parameters of probabilistic-model simulation.

Routing method	virtual-cut-through
Bandwidth of channel	1 flit/clock
Packet length	6 flit
Traffic pattern	uniform random
Packet rate	1%
Routing	deterministic routing
Total steps	100,000

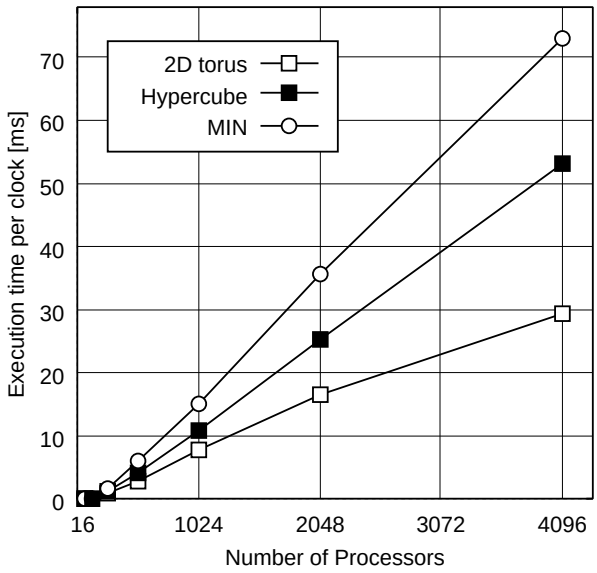

Figure 9. Performance of probabilistic-model simulators.

4.3. Performance of an Instruction-level Simulator

Here a small scale multiprocessor is chosen as a target, and instruction-level simulation scheme is applied. The target multiprocessor provides several processors (MIPS R3000) connected with an ideal shared memory. Applications shown in Table 4 from SPLASH2 programs[11] are used as workloads. The number of processors is changed from 1 to 64.

Figure 8 shows simulation time corresponding 1 clock of the target machine. According to Figure 8, the execution time per step is approximately in proportion to the number of processors, and not much depending on applications. It shows that a simulator for a system with 4 processors can execute 40,000 to 70,000 steps per second. Thus ISIS can generate a high speed simulator sufficient to evaluate the target machine in real researches.

4.4. Performance of Probabilistic-model Simulators

Next target machines are large scale multiprocessors, and probabilistic-model simulation scheme is applied. Two dimensional torus, hypercube or MIN (Multistage Interconnection Network) are used as a network connecting node processors. The node number of the multiprocessors is changed from 16 to 4,096. Rest of parameters are shown in Table 5.

Figure 9 shows that required execution time to process 1 clock of the target machines. Regardless of connection topologies, only about 7 to 15 microsecond is required for an execution of a step of the target machine with 1,024 nodes. Since the routing algorithm is changed with the network topology, the results are slightly different depending on the network size.

Table 6. Breakdown of source codes.

	UNI	MULTI	2D	HC
System model	2,710	2,710	2,467	2,467
Device	1,914	1,914	637	637
Processor	10,683	10,683	-	-
Memory cntl.	543	543	-	-
I/O and etc.	1,460	1,460	-	-
Network I/F	-	-	2,183	2,183
Router	-	-	906	906
PUs manage.	-	617	-	-
Private code	203	756	233	172
Priv./Share.	1.16%	3.98%	3.63%	2.70%

4.5. Implementation Cost

Amount of source codes are counted to evaluate how ISIS can contribute to reduce the cost of simulator implementation. Three types of simulators in this section: an instruction-level uniprocessor simulator "UNI" described in Section 4.2, an instruction-level multiprocessor simulator "MULTI" in Section 4.3, and probabilistic-model multiprocessor simulators "2D"/"HC" in Section 4.4. 2D shows a machine with two dimensional torus while HC shows one with the hypercube interconnection.

Table 6 shows the amount of source code of each part in lines. "System model" means the code of the basic parts of ISIS, e.g. the base classes of *Unit*, *Packet* and *Port*. "Private code" means the written code only for the distinct simulator.

98.84% of the source code of UNI are shared with MULTI. This comes from that they are both instruction-level simulator which uses the same parts.

On the other hand, 2D and HC share few codes with UNI and MULTI except what concerns system model, since they are probabilistic-model simulator. However, 96.37% of the source code of 2D are shared with HC, since their structure is almost same except their connection topologies.

These results show that small codes are only required to described by the user, and a large part of code can be shared with other simulators.

5. Researches with ISIS

ISIS has been developed since 1996 and utilized in several research projects. In this section, some of experiences are introduced.

5.1. pSAS Cache

Inoue et al. investigated trade-off between shared and snoop cache for on-chip multiprocessors[6], and proposed a new cache mechanism called pSAS(Pseudo Set Associative and Shared) cache. In this cache, by using the snoop cache modules attached to other processors in the same chip as extra ways of its own cache, it combines both advantages of snoop and shared cache. Simulation results show that the performance is improved 10% in average and 16% at maximum than a traditional snoop cache mechanism.

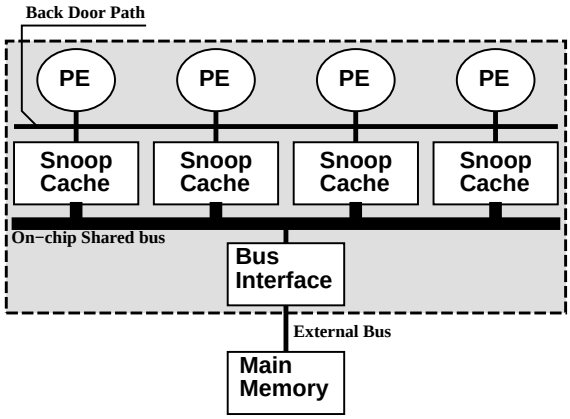

Figure 10. Structure of pSAS cache.

Figure 10 shows the structure of pSAS cache. An extra data path called BDP (Back Door Path) is added to use other cache modules as extra cache sets. In order to save the required hardware, the bandwidth of BDP is assumed to be a word. Access to other cache modules via the BDP takes a few additional clocks, which are much less than latency to access the outside memory. In the case of access conflicting between the access via the BDP and local access, local cache access has a higher priority than access from other processors. A tag memory must have multiple ports and enable to access the cache entry both from the local and the BDP without conflict.

Through the evaluation of pSAS cache, the behavior of processors, caches, buses, and memory module outside the chip are important. Thus, it must be simulated in the clock level with practical parallel application benchmarks. On the other hand, the size of the target system is rather small, since it is assumed to be implemented on a single chip. Therefore, an instruction-level simulator is built with ISIS library.

Figure 10 shows the pSAS and corresponding simulator structure. All the *Units* are included in ISIS library except the cache controllers and the bus interfaces. Derived classes from `port` class and `packet` class are used in order to implement the shared bus. Other simulators for common multiprocessor with snoop cache is also implemented for performance comparison.

Almost four months are spent, since the simulator development started until ISIS had not achieved stable working. 4,000 lines code is required in the simulator for representing the target architecture, and about 22,000 lines had been

already included in ISIS.

5.2. Shared vs. Snoop

Kisuki et al. in Keio University compared the performance of the snoop cache architecture with that of the shared cache architecture on an on-chip multiprocessor[6]. It is realized that a snoop cache has higher performance in consideration of access delay for shared memory, according to the quantitative evaluation with some SPLASH2 programs on the instruction-level simulators of both architecture. These two simulators are implemented with ISIS and this research results became the basis of above pSAS cache.

5.3. Interconnection Network Simulator

ISIS had been used for the evaluation of interconnection network used in a massively parallel computer JUMP-1[3] which was developed with cooperation of seven Universities. In this case, probabilistic-model simulation was used and a network congestion with 1,024 nodes were evaluated[4]. In this project, an instruction-level simulator was also implemented for precise level simulation of a system with 16 processors.

5.4. Other activities with ISIS

Some other projects with ISIS are now in progress: a simulator of a novel interconnection network called Recursive Diagonal Torus[12] is now under development with ISIS. Unlike other interconnection network simulators, it is not a probabilistic-model but an instruction-level simulator which can evaluate detail communication congestion with practical parallel applications. Another instruction-level simulator of MAPLE[2] is also now developing for a custom processor for the multi-grain parallelization multiprocessor ASCA[5].

6. Conclusion

A multiprocessor simulator generation kit called ISIS is proposed, designed and evaluated.

Some multiprocessor simulators with ISIS are implemented and their total simulation time are measured: for an instruction-level simulator of a small scale multiprocessor up to 64 processors, and for a probabilistic-model simulator of the large scale multiprocessor with 4,096 nodes. According to the result, the instruction-level simulator with 4 processors can process 40,000 to 70,000 steps per second, and the probabilistic-model simulator which has 1,024 nodes can process 60 to 130 steps per second. These simulation speed are comparable to the traditional dedicated simulators. Most of source codes can be shared, and private code sizes are about 4% at the most. Therefore it proves that ISIS reduces the implementation cost without increasing the runtime cost.

ISIS is published as a free software since October, 2000. Please See the url: *http://www.am.ics.keio.ac.jp/isis/* .

References

[1] H. Davis, S. R. Goldschmidt, and J. Hennessy. Multiprocessor Simulation and Tracing Using Tango. *Proceedings of International Conference on Parallel Processing*, II:99–107, 1991.

[2] T. Fujiwara, K. Sakamoto, T. Kawaguchi, K. Iwai, and H. Amano. A CUSTOM PROCESSOR FOR THE MULTIPROCESSOR SYSTEM ASCA. In *Proceedings of 16th IASTED International Conference Applied Informatics – AI'98*, pages 258–261, Mar 1998.

[3] T. H. Massively Parallel Processing System JUMP-1. IOS Press, ISBN 90-5199-262-9, 1996.

[4] H. Inoue, K. ichiro Anjo, J. Ymamamoto, J. Tanabe, M. Wakabayashi, M. Sato, H. Amano, and K. Hiraki. The Preliminary Evaluation of MBP-light with two protocol policies for a Massively Parallel Processor JUMP-1. In *Proceedings of Symposium on Frontiers of Massively Parallel Computation*, pages 268–275, 1999.

[5] K. Iwai, T. Morimura, T. Fujiwara, K. Sakamoto, T. Kawaguchi, K. Kimura, H. Amano, and K. Hironori. Interconnection network of ASCA: a multiprocessor for multigrain parallel processing. In *Proceedings of 16th IASTED International Conference Applied Informatics – AI'98*, pages 262–264, Mar 1998.

[6] T. Kisuki, M. Wakabayashi, J. Yamamoto, K. Inoue, and H. Amano. Shared vs. Snoop: Evaluation of Cache Structure for Single-chip Multiprocessors. In *Proceedings of the 3rd International Europian Conference on Parallel Processing – Euro-Par'97*, pages 793–797, Feb 1997.

[7] A.-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The Augmint multiprocessor simulation toolkit for Intel x86 architectures. In *Proceedings of International Conference on Computer Design: VLSI in Computers and Processors*, pages 486–490, Oct 1996.

[8] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology: Systems & Applications*, 3(4):34–43, Winter 1995.

[9] R. L. Sites and A. Agarwal. Multiprocessor Cache Analysis Using ATUM. In *Proceedings of 15th International Symposium on Computer Architecture*, pages 186–195, 1988.

[10] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. volume 24, pages 68–79, May 1996.

[11] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, Jun 1995.

[12] Y. Yang and H. Amano. Message Transfer Algorithms on the Recursive Diagonal Torus. *IEICE Transaction on Information and Systems*, E79-D(2):107–116, Feb 1996.