ISIS: MULTIPROCESSOR SIMULATOR LIBRARY

MASAKI WAKABAYASHI* KEISUKE INOUE*
HIDEHARU AMANO*
*Department of Computer Science, Keio University
3-14-1, Hiyoshi, Kohoku-ku, Yokohama 223-8522 Japan.
{masaki,keisuke,hunga}@am.ics.keio.ac.jp

Abstract

In this paper, architecture independent software simulation kit for multiprocessors called ISIS is proposed and designed. It includes various small simulators of a hardware device. All functions are implemented in C++ language, and the extension of them is also easy.

Execution speed of a sample simulator was measured. Execution time of the simulator with four R3000 processors is 22,000 times of the real hardware (target system is assumed to work with 500MHz system clock). It is reasonable speed to evaluate a real multiprocessor.

KEYWORDS
"multiprocessor", "software simulation", "performance evaluation", "architecture independent", "library"

1 INTRODUCTION

Not only structure of a modern multiprocessor but also data structure and control flow of most parallel applications become complicated. For researchers and designers who need to evaluate such complicated multiprocessor systems, a software simulation is one of the most effective method, since there is no limitation on current device technology nor hardware configuration.

A lot of software simulators have been developed and used[1, 2, 3]. However, when the target system is modified, the simulator must be modified for it. It is sometimes a painful job for researchers to modify such a simulator, especially when the target system is a novel research machine.

In this paper, architecture independent simulation kit for multiprocessors called ISIS is proposed. It includes various functions which can be useful to develop simulators, such as a processor, bus, memory, cache, and some I/O device. Further, this library supports various types of simulation method — instruction level simulation, trace-driven simulation, and probabilistic simulation. All functions are encapsulated into a *unit*, which represents each function block in a real hardware. All *units* can be connected with each other using *packet* and *port* which support connection and communication among *units*. Using ISIS, researchers can build simulators for their original target architectures only by describing their peculiar part of the architecture and connecting them with common units in ISIS. Furthermore, since all *units*, *packets* and *ports* are implemented using inheritance of C++ language, the extension of these functions is also easy.

ISIS has been used in real research projects[4]. Instruction level simulators which used in the research on a structure of cache system for a single-chip multiprocessor was developed based on ISIS. The target system includes R3000 processors and a cache system — snoop cache or shared cache. In this case, the researcher must only implement a new snoop cache controller, shared cache controller and bus bridge.

2 DESIGN

There are three major requirements for ISIS. First, high degree of flexibility is required, that is, ISIS should be independent from both architecture and simulation method. Second, ISIS must reduce code size written by users as possible. Third, simulators developed with ISIS must run with a reasonable execution speed.

2.1 SYSTEM MODELING

In order to be independent from both architecture and simulation method, ISIS takes a library structure, which includes a lot of small simulators called *unit* — such as a processing unit, cache, memory and I/O device. To build a simulator of the specific multiprocessor, researchers only connect required *units* each other in the structure of the target machine. Even if the original function block which is not prepared in ISIS exists in the target machine, researchers can build the simulator by adding a new *unit* and connecting it. Furthermore, by encapsulating each *unit* independently, we can reduce costs of implementation and extension.

To make sure of connectivity among various *units*, *packet* and *port* are introduced.

Packet is an entity of information across a signal line or a network — such as a memory access request on bus, and a flit between interconnection network routers. *Port* is an input/output terminal of connection among *units*. Figure 1 shows a sample of system modeling using *unit*, *packet* and *port*. Each *unit* has some *ports* for connecting each other.

2.2 REDUCING CODE SIZE

In order to reduce the code size written by users, ISIS must provide an easy way to implement new *units* by

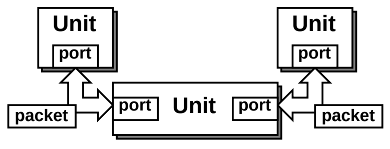

Figure 1: SYSTEM MODELING

users. For this requirement, *units* in ISIS are implemented using a concept of inheritance. If different two *units* have some common functions, their relationship is written by inheritance. A user can easily extend a *unit* in ISIS with inheritance for his own intention.

2.3 EXECUTION SPEED

For reasonable execution speed of simulators, we carefully selected the implementation language of ISIS. Since a software simulation of multiprocessor usually takes a long time, it is mostly implemented with the language which can generate faster code such as C language.

3 IMPLEMENTATION

Considering requirements mentioned before: encapsulation of *units*, the function of inheritance, and a generation of fast simulator code, we select C++ as an implementation language of ISIS. *Unit*, *packet* and *port* are implemented as class, and their relationship is written by class inheritance. That is, ISIS is implemented as a class library.

3.1 UNIT, PACKET, PORT

`synchronous_unit` class is defined as the base class of *units*. All the classes of a *unit* in a multiprocessor are implemented as a derived class of it. `synchronous_unit` class is an abstract class, which presents state transition functions and reset functions to its derived classes. All these functions are virtual functions, so its real processing is implemented in derived classes.

`packet_base` class is defined as the base class of *packets*. All classes of *packet* are implemented as a derived class of it.

`port_base` class is defined as the base class of *ports*. This class provides connect/disconnect functions and send/receive *packet* functions. When some *ports* are connected to each other, the connection path is created automatically. A connection path can pass only one *packet* at a time.

3.2 PROCESSOR CLASS

As a base class of all processors, `processor` class is defined. A processor is also one of a *unit*, and so `processor` class is a derived class of `synchronous_unit` class. It has various virtual functions — accessing to its register file, halt detection, bus access status report, and so on.

As a real processor simulator, `r3000` class which simulates MIPS R3000 processor is implemented in ISIS. To simulate a clock level behavior, it includes register table, 5 stage instruction pipeline, bus controller, instruction/data primary cache, write buffer, system control co-processor, and etc. All these function blocks simulate accurately R3000 processor in pipeline clock level. Using this unit, it is easy to construct the instruction level simulator of a multiprocessor with R3000 processors.

3.3 HOW TO WRITE A SIMULATOR

To implement a simulator for a target machine, following steps are required for users: (1) write a top module using ISIS classes, (2) if new devices are needed, use inheritance from a provided class, (3) compile them, (4) link all with ISIS library. And, the desired simulator will be created (See figure 2).

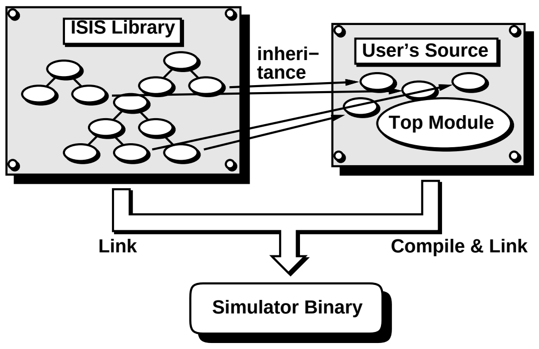

Figure 2: HOW TO USE ISIS

3.4 SAMPLE SIMULATORS

To generate an access trace data, an instruction level multiprocessor simulator called "tracemaker" is provided by ISIS. The structure of it is shown in Figure

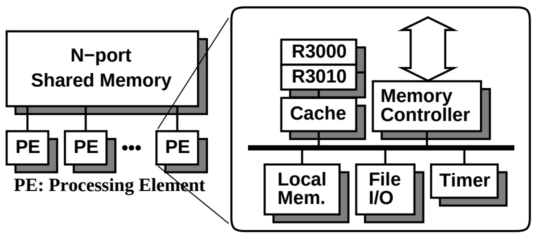

Figure 3: BLOCK DIAGRAM OF TRACEMAKER

3. Each processing element has an R3000 processor, local memory, I/O device, and etc. Its shared memory is "ideal memory" — N-port memory, and it has no

access latency. It is impossible to build a real hardware in this architecture, but it is useful to generate memory access trace file.

4 EVALUATION

First, the execution speed of a simulator using ISIS is measured by simulating the above described trace-maker. The system clock frequency is assumed to be 500MHz, and the processor number is set from 1 to 64. As workloads, some SPLASH2 programs shown in table 1 were used. The simulation host machine is the IBM PC/AT compatible, which has Pentium-II 300MHz and 64MB memory. RedHat Linux-5.2 is used as an operating system.

application	**problem size**
BARNES	512 bodies
FFT	65536 complex doubles
LU	256 × 256 matrix
OCEAN	66 × 66 grid
RADIX	2097152 keys

Table 1: WORKLOAD

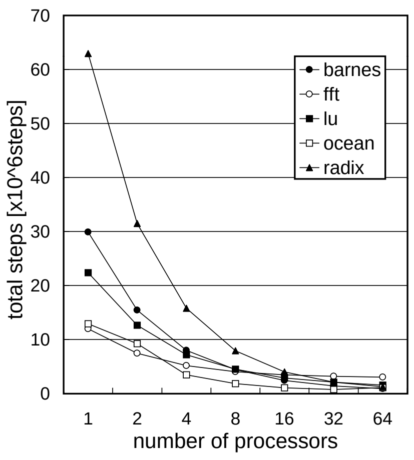

Figure 4: TOTAL STEPS

Figure 4 shows total execution steps. This results shows that these workloads have enough parallelism for estimating performance of a multiprocessor.

Figure 5 shows total execution time. For a single processor system, half hour to two hours were spent. For simulating a system with 64 processors, three hours to eight hours were required. This result demonstrates that the performance evaluation of medium scale multiprocessors will finish in several hours. For a system with 4 processors, execution speed of the simulator is about 22,000 times as the real hardware.

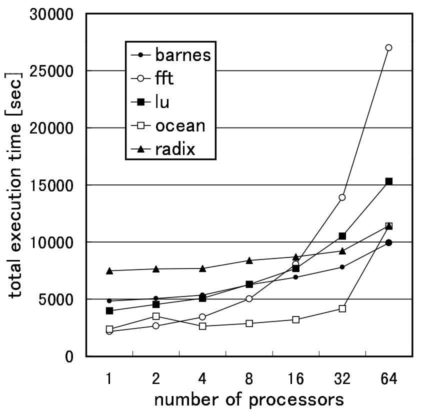

Figure 5: TOTAL TIME

5 CONCLUSION

An architecture independent software simulation kit for multiprocessors called ISIS is proposed and designed. ISIS consists of three elements: *unit*, *packet* and *port*, and they are implemented as C++ classes.

An execution speed of a sample simulator was measured, and it is confirmed that an instruction level simulator using ISIS has reasonable speed to simulate a medium scale multiprocessor.

Now, we are developing some simulators and libraries for new architectures — on-chip multiprocessor, and a large scale NUMA/NORA architecture connected with various interconnection networks.

References

[1] Richard L. Sites, Anant Agarwal, Multiprocessor Cache Analysis Using ATUM, *Proceedings of 15th International Symposium on Computer Architecture*, 1988, pp.186–195.

[2] Helen Davis, Stephen R. Goldschmidt, John Hennessy, Multiprocessor Simulation and Tracing Using Tango, *Proceedings of International Conference on Parallel Processing*, 1991, pp.II-99-II-107.

[3] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, Anoop Gupta, Complete Computer System Simulation: The SimOS Approach, *IEEE Parallel and Distributed Technology*, Fall 1995.

[4] Toru Kisuki, Masaki Wakabayashi, Junji Yamamoto, Keisuke Inoue, Hideharu Amano, Shared vs. Snoop: Evaluation of Cache Structure for Single-chip Multiprocessors, *Euro-Par '97.*

[5] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, Anoop Gupta, The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995, pp 24–36.