The SimCore/Alpha Functional Simulator

Kenji Kise[†,‡], Takahiro Katagiri[†,‡], Hiroki Honda[†], and Toshitsugu Yuba[†]
[†] Graduate School of Information Systems
The University of Electro-Communications
[‡] PRESTO, Japan Science and Technology Agency (JST)
{kis, katagiri, honda, yuba}@is.uec.ac.jp

Abstract

We have developed a function-level processor simulator, SimCore/Alpha Functional Simulator Version 2.0 (SimCore Version 2.0), for processor architecture research and processor education. This paper describes the design and implementation of SimCore Version 2.0. The main features of SimCore Version 2.0 are as follows: (1) It offers many functions as a function-level simulator. (2) It is implemented compactly with 2,800 lines in C++. (3) It separates the function of the program loader. (4) No global variable is used, and so it improves the readability and function. (5) It offers a powerful verification mechanism. (6) It operates on many platforms. (7) Compared with sim-fast in the SimpleScalar Tool Set, SimCore Version 2.0 attains a 19% improvement in simulation speed.

1 Introduction

Various processor simulators [1, 2] are used as tools for processor architecture research or processor education. The environment in which a processor simulator can perform is improving dramatically due to the increased speed of PCs and the growing use of PC clusters. However, the time needed for simulator construction increases as the architectural idea to be implemented increases in complexity. In many cases the evaluation with a simulator can be finished in several weeks, although several months are needed for the construction of the simulator, even if the simulator is developed with existing tools.

SimpleScalar Tool Set [3] and SPIM [4] are well-known processor simulators used for such purposes as processor research and education. But, since SimpleScalar can be implemented as a high-speed simulation, it is not a code that can easily be modified. Similarly, SPIM may not be readable [1]. Although there has been much research on processor simulator speedup [5], there are few simulators which make readability a priority.

[1] We say readable to mean *enjoyable and easy to read.*

In addition to its utilization in processor research and education, a processor simulator is vital as a module in a parallel computer simulator, embedded system emulator, and so on. In these various uses, in addition to high speed, a process simulator must have high readability and be easy to use.

We previously constructed an Alpha [6] processor simulator named SimAlpha [7]. Its design policy was to keep the source code readable and simple. This paper describes the design and implementation of a new generation of processor simulator named SimCore/Alpha Functional Simulator Version 2.0 (SimCore Version 2.0) derived from SimAlpha. SimCore Version 2.0 is a function-level simulator, which satisfies the requirements of high readability and high-speed execution simultaneously. In general, function-level simulators cannot be used to measure processor performance. However, it can be used to collect basic data, such as the number of executed instructions, the hit ratio of a branch prediction and cache, and the ideal instruction level parallelism of a program. It is also vital as a base for preliminary evaluation of detailed clock-level simulation and as a module for verification.

The rest of this paper is organized as follows. Section 2 describes the design and implementation of SimCore. Section 3 reports our quantitative evaluation results. Section 4 is a discussion of related works. Section 5 contains some concluding remarks.

2 Design and implementation

SimCore Version 2.0 features the addition of various functions, an increased number of platforms, and simulation speedup, while inheriting the high readability of Version 1.0. In this section, the function of SimCore is summarized, then the design and implementation issues are discussed.

2.1 Functions

SimCore offers many functions as a function-level or instruction-level simulator. Except for the fact that the tar-

```
 1  # SimCore-Loader a.out > aout.txt
 2  # SimCore aout.txt
 3  SimCore/Alpha Functional Simulator Ver 2.0
 4  hello, world
 5
 6  ==========================================
 7  == SimCore Version 2.0 2004-01-08
 8  == 0 million code ( 2070 code) 0.019 MIPS
 9  == SimCore takes 0 min 0 second(109375 usec)
10  == SimCore starts at Wed Apr 7 18:34:20 2004
```

Figure 1. A sample output of the "hello, world" program.

```
 1    lines   words   bytes  filename
 2       19      54     525  sim.cc
 3       75     188    2067  chip.cc
 4      231     833    6833  instruction.cc
 5      297     872    7221  memory.cc
 6      468    1711   12817  arithmetic.cc
 7      168     401    3915  debug.cc
 8      578    1100   12783  syscall.cc
 9      378    1021    9865  etc.cc
10      550    1724   16565  define.h
11     2764    7904   72591  ---- total ----
```

Figure 2. The number of lines, words and bytes of SimCore code.

get architecture is fixed to the Alpha processor, it has almost the same function as sim-fast in SimpleScalar.

A fundamental function is to simulate an application program for each instruction. As an example, the commands to simulate the well-known program which prints ”hello, world” and its output are shown in Figure 1. Part of the output is formatted so that it is easy to see. The 1st line is a command to translate the application program into a SimCore input file (see Section 2.3). The 2nd line is the command to start SimCore. The 4th line is the output of the application program. The 6th through 10th lines are a report of the simulation. In this example, 2,070 instructions were executed and 109,375 μsec was required for the simulation.

Other implemented functions are enumerated below:

- The function to measure the frequency of appearance of the executed instructions (instruction mix).
- The function to display the history of the executed system calls.
- The interactive debugging function to display the contents of the memory or register at a specified time.
- The debugging support function to output the contents of the main registers for all executed instructions.

2.2 Compact description

A compact description was one goal of the design policy of SimCore. SimCore is written in C++ and the code size containing the include file is small at **2,764 lines**. The number of lines of each file is summarized in Figure 2. Because functions are somewhat different, a direct comparison cannot be made, but the number of lines of other simulators are given for reference. The code for compiling sim-fast in SimpleScalar is 15,566 lines. The code of SPIM is 14,213 lines.

The main reason why such a compact description was attained is that the fundamental software architecture differs from a conventional simulator design.

The part of the main loop in SimpleScalar which processes one instruction is shown in Figure 3. Only a portion

```
 1  switch (op){
 2   case ADDQ:
 3     regs.regs_R[(inst & 0x1f)] =
 4       regs.regs_R[(inst >> 21) & 0x1f] +
 5       regs.regs_R[(inst >> 16) & 0x1f];
 6     break;
 7   case MULQ:
 8     regs.regs_R[(inst & 0x1f)] =
 9       regs.regs_R[(inst >> 21) & 0x1f] *
10       regs.regs_R[(inst >> 16) & 0x1f];
11     break;
12  }
```

Figure 3. The main loop implementation of sim-fast.

of an add instruction (ADDQ) and a multiply instruction (MULQ) is shown.

The type of instruction to be processed is distinguished in the switch sentence of the 1st line in Figure 3. Then, the block from the case sentence of the 2nd line through the 6th line describes the operation of addition. The sum of the value of the registers is calculated in the 4th and 5th lines, and the result is stored in a register in the 3rd line. The block from the 7th line through the 11th line describes the operation of multiplication. This style, which describes the operation of each instruction independently, is named an **unfolded description style**. This style permits an easy change of operation of an instruction, and the addition of a new instruction. Since the processing which is necessary for each instruction is described, there is the advantage that a high-speed simulation is possible. On the other hand, because the same description appears in two or more parts, there are the drawbacks that management of the source code becomes complicated and the amount of code becomes large.

In order to remove these drawbacks, a style can be considered which extracts the common part of the operation of each instruction and describes its operation gradually with reference to the pipeline structure of a microprocessor. We name this style a **folded description style**.

SimCore adopts a folded description style. The code of the method step which processes one instruction in SimCore

```
 1  int simple_chip::step(){
 2    p->Fetch(&ev->as->pc); /* pipe stage 0 */
 3    p->Slot();             /* pipe stage 1 */
 4    p->Issue();            /* pipe stage 3 */
 5    p->RegisterRead();     /* pipe stage 4 */
 6    p->Execute();          /* pipe stage 5 */
 7    p->Memory();           /* pipe stage 6 */
 8    p->WriteBack();
 9    return ev->sys->running;
10  }
```

Figure 4. The main loop implementation of SimCore. SimCore adopts a folded description style.

```
1  /* SimCore 1.0 Image File */
2  /*** Registers ***/
3  /@reg 16 0000000000000003
4  /@pc  32 0000000120007d80
5  /*** Memory    ***/
6  @11ff97000 00000003
7  @11ff97008 1ff97138
```

Figure 5. A sample SimCore execution image file.

is shown in Figure 4. The class instruction which holds the information for processing one instruction is defined. The p in Figure 4 is the object of the class instruction. The values of the private variables are gradually determined by calling the method (Fetch, Slot, Issue, RegisterRead, Execute, Memory, or WriteBack) of object p which corresponds to the pipeline stage.

A folded description style is close to the description of the microprocessor implemented in a hardware description language such as verilog-HDL. For this reason, the advantage is that the operation of the hardware can be easily captured. Since the common operation is described in one place, management of the code becomes easy. On the other hand, compared with the unfolded description style which describes the operation of each instruction independently, the folded description style is a disadvantage with respect to extendibility and simulation speed.

2.3 SimCore and program loader

SimCore does not have the function of a program loader, which may be regarded as an additional function of a processor simulator.

A simulation is started using the execution image of the original format. An example of the execution image file of SimCore is shown in Figure 5. This execution image file is in text format and consists of two parts. In the first part, values are assigned to some of the registers. In the example of Figure 5, the hexadecimal value 3 is assigned to the 16th register, and the value of 120007d80 is assigned to the program counter. In the second part, the value of some memory is assigned in the same manner. In the example of Figure 5, the value 1ff97138 is assigned to the memory of address 11ff97008.

The execution image file is created from the Alpha binary file with a program named SimCore-Loader [8]. As an example, the command to simulate li (lisp interpreter) from SPEC CINT95 on sim-fast of SimpleScalar is shown.

```
1  $ sim-fast li train.lsp
```

The corresponding command of SimCore is shown. The command in the 1st line generates the execution image file. The simulation is started by the command in the 2nd line.

```
1  $ SimCore-Loader li train.lsp > aout.txt
2  $ SimCore aout.txt
```

The simulation of an identical application is repeated many times with various simulation parameters. In SimCore, once an execution image file is created, only the name of the execution image file is specified to run the simulator. This mitigates any mistake at the time the simulation starts. Moreover, in the utilization of SimCore, knowledge of the executable file form of ELF or COFF[9] is not required. This is an advantage because users can concentrate on the description of the behavior of the processor simulator.

2.4 Elimination of global variables

No global variable is used in the SimCore code. On the other hand, many global variables are used in SimpleScalar and SPIM.

The readability of the source code is improved by eliminating global variables. In addition, elimination of global variables is important from the viewpoint of the function of a simulator. For example, let's consider the measurement of the branch prediction and the cache behavior when switching two or more tasks or threads in a processor. In this case, it is necessary to switch the application at a fixed interval. Since SimCore does not use a global variable, it is possible to describe such behavior compactly.

The main function to switch two applications with a 5,000-instruction interval is shown in Figure 6. In the 5th line and 6th lines, the simple_chip type objects p1 and p2 are generated. Then, 5,000 instructions of one application are processed by the for loop in the 9th line. The method step called in the 9th line is described in Figure 4. Similarly, 5,000 instructions of another application are processed by the for loop in the 10th line. It becomes possible to measure the behavior of the branch prediction or cache in the task-switching environment by inserting a branch or cache module in the code shown in Figure 6.

As shown in Figure 6, SimCore can generate two or more simulation images with a compact description. This makes it easy to use SimCore as a module of a complicated computer system or a parallel computer system. In contrast, it is difficult to generate two or more simulation images in a

```
 1  int main(int argc, char *argv[]){
 2    char *p1 = argv[argc-1]; /* program name*/
 3    char *p2 = argv[argc-2]; /* program name*/
 4
 5    simple_chip *c1=new simple_chip(p1, argv);
 6    simple_chip *c2=new simple_chip(p2, argv);
 7
 8    for(int i=0; i<100; i++){
 9      for(int j=0; j<5000; j++) c1->step();
10      for(int j=0; j<5000; j++) c2->step();
11    }
12
13    delete c2;
14    delete c1;
15    return 0;
16  }
```

Figure 6. The main function, which switches two applications with a 5,000-instruction interval.

single process with a conventional simulator implemented using global variables.

In addition to the elimination of global variables, in order to make it readable, neither goto statements nor conditional compilation is used.

2.5 Verification feature

A processor simulator is complicated software and there is the possibility that it may have various bugs. At the time of development, sufficient verification is necessary.

In conventional simulators, it was difficult to make two or more simulation images in one process due to the existence of global variables. For this reason, the correct execution result was saved to a file and another simulator was verified by comparison with this file. As a means of high-speed verification, SimCore offers a function which embeds the object of SimCore for another simulator such as SimpleScalar. We offer the C language interface to realize this feature.

At the time of development of SimCore Version 2.0, the behavior was verified using this verification feature. Whenever the simulator executed one instruction, all values of the architecture state (a program counter, 32 integer registers, 32 floating point registers) of SimCore and the architecture state of SimpleScalar were compared and we confirmed that the two architecture states were identical during the 20 benchmark simulations of SPEC CINT95 and CINT2000.

Two or more simulation images can easily be generated in one process, as shown in Figure 6. By using this feature, any bug of new simulators under development is discovered at an early stage. Also, by using the feature, developers can prove the validity of their simulator.

2.6 Platforms

SimCore Version 2.0 operates on more platforms than did the previous version. The platforms where the correctness of operation has been verified are enumerated. On seven platforms, operation has been verified with dhrystone and 20 programs from SPEC CINT95 and CINT2000.

- Pentium 4, RedHat 7.3, GCC version 2.96
- Pentium 4, RedHat 7.3, Intel C++ 7.1/8.0
- Pentium 4, RedHat 7.3, PGI Compiler 5.1
- Pentium 4, Cygwin version 2.340, GCC 3.2
- Pentium 4, FreeBSD 4.9, GCC 2.95.4
- Opteron, Turbo Linux 8, GCC version 3.2.2
- Alpha 21264, Tru64, GCC version 2.95.2

On two platforms, operation has been verified with dhrystone.

- UltraSPARCIII, Solaris, GCC version 2.95.3
- MIPS R14000, IRIX6.5, MIPSpro C++

SimCore operates on these major platforms. Because a processor simulator is used in various environments, it needs to support many platforms. In contrast, SimpleScalar Version 3.0 has not been compiled with either an Intel compiler or a MIPSpro compiler.

2.7 Simulation speedup

This section discusses the tuning technique implemented in the main loop of SimCore while keeping the high readability of the source code.

2.7.1 Pipeline frontend reuse

A simple main loop without optimization is shown in Figure 7. As discussed in Section 2.2, one instruction is executed by calling seven methods corresponding to the instruction pipeline from the 3rd line through the 9th line of Figure 7.

Three methods, Fetch, Slot and Issue, in the simple main loop in Figure 7 take charge of fetching the 32-bit instruction code, decoding, and calculating an immediate value, respectively. These methods correspond to the pipeline frontend. If the simulator is processing the statically same instruction, identical processing is repeated each time in these methods. Therefore, the calculation result obtained is saved in memory and it is possible to improve the simulation speed using the calculation result. This speedup technique is called pipeline frontend reuse.

The main loop with the pipeline frontend reuse is shown in Figure 8. The pointer array ib of the type instruction (the 3rd line), which contains the past calculation result,

```
 1  void simple_chip::loop_simple(){
 2    while(ev->sys->running){
 3      p->Fetch(&ev->as->pc); /* pipe stage 0*/
 4      p->Slot();             /* pipe stage 1*/
 5      p->Issue();            /* pipe stage 3*/
 6      p->RegisterRead();     /* pipe stage 4*/
 7      p->Execute();          /* pipe stage 5*/
 8      p->Memory();           /* pipe stage 6*/
 9      p->WriteBack();
10
11      ev->e->retired_inst++;
12      house_keeper(p);
13    }
14  }
```

Figure 7. A simple implementation of the SimCore main loop.

is prepared in the same way as the direct-mapped instruction cache. The number specified by the constant IMSK is the number of entries of the array, and is set as a 64K entry. The index of the array is generated from the program counter (the 9th line). If the program counter from the array (the 10th line) and from the instruction currently executed differs (the 12th line), Fetch, Slot, and Issue (from the 13th line through the 15th line) are executed. Otherwise, the past history is used and the pipeline frontend (from the 13th line through the 15th line) is omitted.

In the method implemented in Figure 8, the rate at which the pipeline frontend can be omitted is the same as the high hit ratio of the direct-mapped instruction cache of the 64K entry. Therefore, in the execution of most instructions, it is possible to eliminate the processing of the pipeline frontend for Fetch, Slot, and Issue.

This method is not new in software engineering. However, the fact that this technique can be implemented without lessening readability is important.

2.7.2 Function call overhead elimination

By adopting pipeline frontend reuse, most of the execution time of SimCore is spent in the pipeline backend. At this time, the function call overhead in the 17th line through the 20th line in Figure 8 becomes notable. In order to reduce this overhead, processing of the four methods which organize the pipeline backend is described as one method, named BackEnd. The code after elimination of the function call overhead, which corresponds to the main loop from the 8th line through the 24th line in Figure 8, is shown in Figure 9. The code is replaced with the method BackEnd in the 10th line.

In SimCore Version 2.0, the main loop shown in Figure 9 is adopted in order to simultaneously attain a compact description and an improvement in speed.

```
 1  #define IMSK 0x0ffff  /* mask of inst_buf */
 2  void simple_chip::loop_reuse(){
 3    instruction **ib=new instruction*[IMSK+1];
 4    for(int i=0; i<IMSK+1; i++){
 5      ib[i] = new instruction(ev);
 6    }
 7
 8    while(ev->sys->running){
 9      int index = (ev->as->pc>>2) & IMSK;
10      instruction *pt = ib[index];
11
12      if(pt->Cpc!=ev->as->pc){
13        pt->Fetch(&ev->as->pc);
14        pt->Slot();
15        pt->Issue();
16      }
17      pt->RegisterRead();
18      pt->Execute();
19      pt->Memory();
20      pt->WriteBack();
21
22      ev->e->retired_inst++;
23      if(ev->sc->slow_mode) house_keeper(pt);
24    }
25  }
```

Figure 8. A main loop with pipeline frontend reuse.

3 Evaluation of simulation speed

In this section, as a quantitative evaluation of SimCore, the simulation speed of SimCore Version 2.0 is measured and compared with sim-fast in SimpleScalar. Moreover, the influence of the technique discussed in Section 2.7 is examined.

A total of eight benchmark programs from SPEC CINT95 are used to evaluate the simulation speed of SimCore. An input parameter is adjusted so that the number of simulated instructions is reduced from about 100 million to 200 million instructions. The binary of the benchmark programs is generated using a DEC C compiler with the optimization option O4.

The data in this section is measured using a PC with two Pentium4 Xeon 2.8 GHz processors and 2 GB main memory running RedHat Linux 7.3.

3.1 Simulation speed comparison

In this section, the simulation speed of SimCore is measured and compared with sim-fast in SimpleScalar. For the evaluation measure, the number of instructions processed per second (MIPS: Million Instructions processed Per Second) is used.

The evaluation result is summarized in Figure 10. The x-asis indicates benchmark names and the average of 8 benchmark programs. SimCore and sim-fast are compiled using GCC with the optimization option O3. The SimCore simulation speed is faster than sim-fast in all of the bench-

```
 1    while(ev->sys->running){
 2      int index = (ev->as->pc>>2) & IMSK;
 3      instruction *pt = ib[index];
 4
 5      if(pt->Cpc!=ev->as->pc){
 6        pt->Fetch(&ev->as->pc);
 7        pt->Slot();
 8        pt->Issue();
 9      }
10      pt->BackEnd();
11
12      ev->e->retired_inst++;
13      if(ev->sc->slow_mode) house_keeper(pt);
14    }
```

Figure 9. A main loop with pipeline frontend reuse and the elimination of function call overhead.

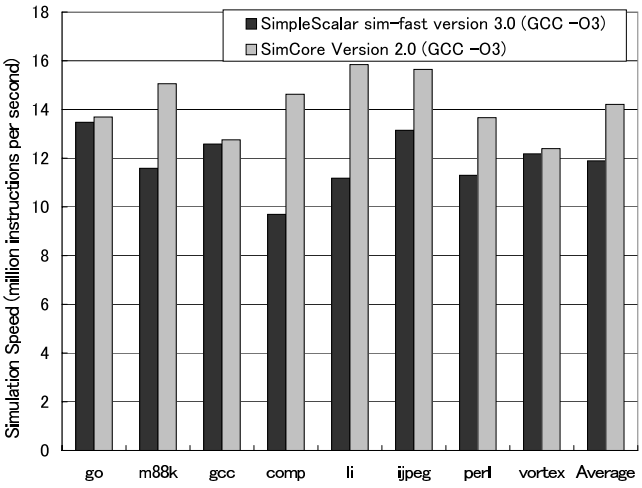

Figure 10. Simulation speeds of sim-fast and SimCore.

mark programs. In particular, in the simulation of compress (comp), SimCore attains the highest speed improvement, which is 50%. In the average of the eight benchmarks, the simulation speed of SimCore is 14.2 MIPS. A speed improvement of 19% is attained compared to the 11.9 MIPS simulation speed of sim-fast.

Next, the simulation speed measured with various compilers and optimization flags are summarized in Figure 11. Five sets of data are shown in this figure for each benchmark. The 1st and 2nd bars from the left are the same data shown in Figure 10.

The 3rd from the left is the data using the Intel C++ Version 7.1 compiler (icc) with optimization option O3 and optimization between files. The simulation speed in this case is an average of 15.3 MIPS. Sim-fast was not able to be compiled using the Intel C++ compiler. SimCore compiled using Intel C++ Version 7.1 compiler with optimization option O3 attains a 28% speed improvement compared with sim-fast compiled using GCC.

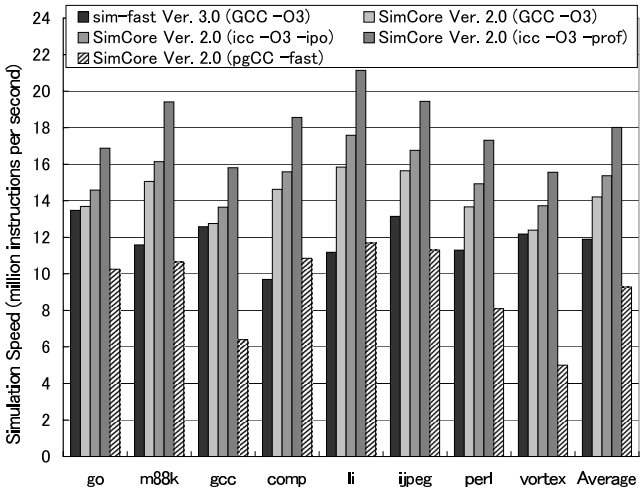

Figure 11. Simulation speed measured with various compilers and optimizations.

The 4th from the left is the data adding the optimization with profile information. As the profile data, the execution history of the dhrystone of a 10,000-times loop is used. The compile time including the execution time for acquiring this profile is very short at less than 5 seconds. The simulation speed becomes an average of 18.0 MIPS by using the profile optimization of the Intel C++ compiler. In this configuration, SimCore attains a 51% speed improvement compared with sim-fast.

The 5th data (on the right end) is the result of using a commercial PGI Compiler 5.1 (pgCC) with the -fast option. The simulation speed in this case is an average of 9.3 MIPS and is slower than the case using GCC.

From the evaluation results summarized in Figure 11, we confirm that when GCC is used, SimCore attains a 19% speed improvement compared with the simulation speed of sim-fast. When the profile information and the Intel C++ compiler is used, SimCore attains a 51% speed improvement.

3.2 Influence of the tuning methods

In this section, the influence of the tuning methods discussed in Section 2.7 is evaluated quantitatively.

The simulation speed of SimCore with versions of the main loop is summarized in Figure 12. The simulation speed shown here is the average of the eight benchmarks of SPEC CINT95.

The uppermost data in Figure 12 is the simulation speed with the main loop of the simple implementation shown in Figure 7. The simulation speed of this version of SimCore is 7.1 MIPS.

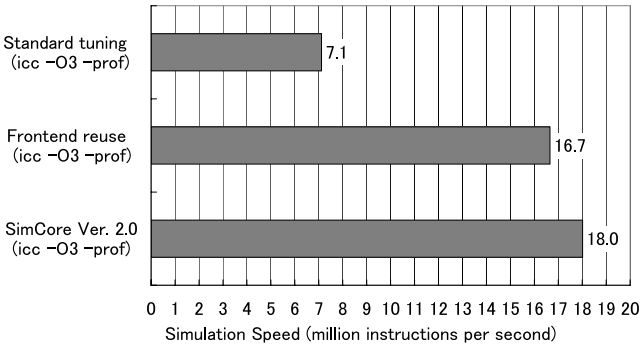

Figure 12. Influence of the tuning methods on SimCore.

The 2nd data is the simulation speed of SimCore with the main loop shown in Figure 8, in which pipeline frontend reuse is used. The simulation speed of this version is 16.7 MIPS. A speed improvement of more than twice is attained by using this technique.

The 3rd data is the simulation speed of SimCore Version 2.0 (Figure 9), in which pipeline frontend reuse and function call overhead elimination are used. The function call overhead elimination brings about a 7% improvement in speed compared to the 2nd data. The simulation speed of SimCore Version 2.0 reaches 18.0 MIPS.

As shown with these results, pipeline frontend reuse and function call overhead elimination attain the highest improvement in simulation speed, even though these are techniques which can be implemented compactly.

4 Related work

Various processor simulators are used as tools for processor architecture research or processor education. In addition, the demand for a faster processor simulator has been growing in recent years with the diversification of processors, including reconfigurable devices such as FPGAs. The focus of some research [10, 11] has been on speed improvement of various instruction set architectures. However, SimCore avoids complicating the source code in order to support many instruction sets. One of the features of SimCore is that it operates as a practical simulator with about 2,800 lines of code, which is a small amount.

There is much research on improving the speed of the processor simulator. In some research [12, 13], speed improvement is attained by lowering the accuracy of the simulation. However, the lower simulation accuracy makes verification difficult. Therefore, lessening the simulation accuracy is not allowed in the design policy of SimCore.

Concerning the speed improvement using reuse or memorization, a FastSim simulator [5] and scheduling calculation reuse have been proposed. In SimCore, pipeline frontend reuse is used as a technique which is realized with little changing of the code. Although this technique is not new, one of the features of SimCore is that techniques are selected in order to prioritize code readability and compact implementation.

SimpleScalar Tool Set [3] and SPIM [4] are well-known processor simulators used for purposes such as processor research and education. But, since SimpleScalar can be implemented in high-speed simulations, it is not a code that can easily be modified. Similarly, SPIM cannot be said to be readable. On the other hand, SimCore Version 2.0 satisfies the requirements of high readability and high-speed execution at the same time.

Historically, the development of SimCore for the C version began in March, 1999. Development of SimCore for the C++ version began in June, 1999. A processor simulator is an important tool, and it is advantageous to choose the most suitable tool, given many choices. As a tool for processor research and education, SimCore offers another choice.

5 Conclusions

We have developed a function-level processor simulator, SimCore/Alpha Functional Simulator Version 2.0 (SimCore Version 2.0), for processor architecture research and processor education. It satisfies the requirements for high readability and high-speed execution at the same time.

In this paper, we discussed the design and implementation of SimCore Version 2.0 in detail. The main features of SimCore Version 2.0 are as follows: (1) It offers many functions as a function-level simulator. (2) It is implemented compactly with 2,800 lines in C++. (3) It separates the function of the program loader. (4) Global variables are not used in order to improve the readability and function. (5) It offers a powerful verification mechanism. (6) It operates on many platforms. (7) Compared with sim-fast in the SimpleScalar Tool Set, SimCore Version 2.0 attains a 19% improvement in simulation speed.

For quantitative evaluations with SPEC CINT95 benchmarks, the simulation speed of SimCore Version 2.0 was measured and compared with sim-fast in the SimpleScalar Tool Set. We confirmed that when GCC is used, SimCore attains a 19% speed improvement compared with the simulation speed of sim-fast. And, when profile information and the Intel C++ compiler are used, SimCore attains a 51% speed improvement.

SimCore/Alpha Functional Simulator Version 2.0 is free software. The source code is downloadable from the following URL.

`http://www.yuba.is.uec.ac.jp/~kis/SimCore/`

References

[1] Shubhendu S. Mukherjee, Sarita V. Adve, Todd Austin, Joel Emer, and Peter S. Magnusson. Perfor-

mance simulation tools. *IEEE Computer*, 35(2):38–39, 2002.

[2] The microlib project. http://www.microlib.org/.

[3] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.

[4] David A. Patterson and John L. Hennessy. *Computer organization and design the hardware/software interface*. Morgan-Kaufmann Publishers, 1998.

[5] Eric Schnarr and James R. Larus. Fast out-of-order processor simulation using memoization. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 283–294, 1998.

[6] R. E. Kessler. The alpha 21264 microprocessor. *IEEE Micro*, 19(2):25–36, 1999.

[7] Kenji Kise, Hiroki Honda, and Toshitsugu Yuba. simalpha version 1.0: simple and readable alpha processor simulator. *Lecture Note in Computer Science (LNCS)*, 2823:122–136, September 2003.

[8] Kenji Kise, Hiroki Honda, and Toshitsugu Yuba. Implementation of simalpha-loader and construction of cross-development environment. Technical Report UEC-IS-2003-5, Graduate School of Information Systems, The University of Electro-Communications, 2003.

[9] John R. Levine. *Linkers and loaders*. Morgan-Kaufmann Publishers, 1999.

[10] Wai Sum Mong and Jianwen Zhu. A retargetable micro-architecture simulator. In *Proceedings of the 40th conference on Design automation*, pages 752–757, 2003.

[11] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *Proceedings of the 39th conference on Design automation*, pages 22–27, 2002.

[12] Bob Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 128–137, 1994.

[13] Thomas M. Conte, Mary Ann Hirsch, and Kishore N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 International Conference on Computer Design (ICCD)*, 1996.