Regular Paper

SimCell: A Processor Simulator for Multi-Core Architecture Research

Shimpei Sato,[†1] Naoki Fujieda,[†1] Akira Moriya[†1] and Kenji Kise[†1]

We developed a new open source multi-core processor simulator SimCell from scratch. SimCell is modeled around the Cell Broadband Engine. In this paper, we describe the advantages of the functional level simulator SimCell. From the verification of the simulation speed, we confirm that SimCell achieves a practical simulation speed. And, we show the features of a cycle-accurate version of SimCell called SimCell/CA (CA stands for cycle accurate). The gap of execution cycles between SimCell/CA and IBM simulator is 0.8% on average. Through a real case study using SimCell, we clarify the usefulness of SimCell for processor architecture research.

1. Introduction

Multi-core processors like Cell Broadband Engine [1),2)] and Intel Core 2 Duo are becoming common in high-performance servers and desktop computers. Moreover, they are used to realize complicated information processing for embedded systems such as home appliances.

Research on computer architectures must investigate various configurations for the next generation of processors. For the research, processor simulators are fundamental software necessary for the software development of new generation of processors.

When we started a multi-core research project in October 2006, we were unable to find a simple and attractive multi-core processors simulator. Therefore, we began the development of the new multi-core processor simulator SimCell from scratch. SimCell is modeled around the Cell Broadband Engine (Cell/B.E.). **Figure 1** shows the block diagram of Cell/B.E. architecture. PowerPC Processor

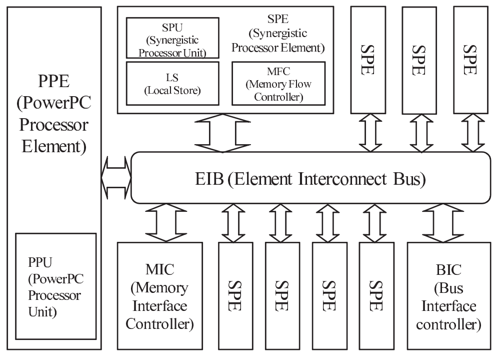
PPE (PowerPC Processor Element)
PPU (PowerPC Processor Unit)
SPU (Synergistic Processor Unit)
LS (Local Store)
SPE (Synergistic Processor Element)
MFC (Memory Flow Controller)
SPE
SPE
SPE
EIB (Element Interconnect Bus)
MIC (Memory Interface Controller)
SPE
SPE
SPE
SPE
BIC (Bus Interface controller)

Fig. 1 The block diagram of Cell/B.E. architecture.

Element (PPE) is a general purpose processor. Synergistic Processor Element (SPE) is an original SIMD style processor [3)]. They are all connected by a ring bus named Element Interconnect Bus (EIB).

In this paper, first, we describe the advantages of the functional level simulator SimCell. Here, the functional level simulator refers to a simulator adopting the execution model of a simplified processor that executes one instruction per cycle. In this kind of simulator, we cannot observe the detailed processor behavior, such as the pipelining, the caching, or the branch prediction. However these simulators are useful for evaluating basic data such as the distribution of the executing instruction (instruction mix), the memory access pattern, and the communication pattern. In addition, functional level simulators are important because they provide a base for complicated simulators modeling a processor in detail.

Second, as a real case study using SimCell, we describe the evaluation of the new software cache implementation on the SPE. In this implementation, we use two SPEs. One is used to run the application program, and the other is used to run the L2 software cache program. This mechanism is originally proposed in our paper [4)]. In order to verify the effectiveness of our software cache, we evaluate the performance changing the on-chip communication latency (cycles required to communicate between two SPEs). In an actual computer, it is very hard to change the communication latency. Through this case study, we clarify the usefulness of SimCell for processor architecture research.

†1 Graduate School of Information Science and Engineering, Tokyo Institute of Technology

Third, we show the features of a cycle-accurate version of SimCell called SimCell/CA (CA stands for cycle accurate). This simulator is implemented by the addition of approximately 160 lines to the functional level version. Through the comparison of SimCell/CA and the IBM simulator[5] and through the verification, we demonstrate the high accuracy of SimCell/CA. In order to demonstrate its effectiveness, we describe the performance of SPE with a hardware branch predictor. This evaluation is originally reported in our paper[6].

Our SimCell paper was published in Refs. 7), 8) where the early version of SimCell was introduced. This paper is a revised version of Ref. 9).

2. Related Works

As of October 2006, we could find no simple and suitable simulator of multi-core processor.

As a multi-core or multiprocessor emulator using FPGA, the RAMP project[10] is attractive. Their approach using FPGA has an advantage in attaining the high-speed simulation. However, it has the disadvantage that special hardware is required. Our approach uses only software-based simulators and provides a flexible tool.

When building such a multi-core processor simulator, there is an alternative to implement it from conventional simulators. It is not easy to rebuild conventional processor simulators such as SimpleScalar[11], M5[12], and Simics[13], as a Cell/B.E. model or as an original model simulator.

Although the IBM Full-System simulator[5] is well constructed, it is not adequate for us to use as an infrastructure of processor architecture research because its source code is not available.

Another Cell/B.E. simulator, CellSim[14],[15], was released by Alejandro Rico et al. at the Technical University of Catalonia and Barcelona Supercomputing Center. Unfortunately the details of this simulator were not clear at the initial stage of our development. Because of these constraints, we decided to develop another Cell/B.E. functional simulator named SimCell from scratch.

Our simulator has many advantages compared to CellSim. It is written with only about 6,000 lines of code and is implemented to be simple and readable. These features are inherited from SimCore/Alpha simulator[16]. The code is carefully written to be easy to understand and easy to extend. This high extendibility facilitates the building of new simulators such as a cycle accurate version simulator discussed in Section 5.

3. SimCell

3.1 What is SimCell?

SimCell is a simple and practical functional level simulator of the Cell/B.E. model. We developed SimCell to be used not only as a Cell/B.E. simulator, but also as an infrastructure for various multi-core processors research. Therefore, we aim to evaluate various processor configurations including on-chip network parameters.

In addition, our target is not only the processor architecture researchers but also application programmers. We attempt to provide a practical simulator for programmers that evaluate the performance or visualize the behavior of parallel applications.

3.2 Source Code Simplicity

In developing SimCell, we adopted the following two design requirements:

(1) Maintain the simplicity of coding and adopt object-oriented programming in order to understand source code intuitively.

(2) Rather than excessive optimization for high-speed simulation, we attempted to achieve a practical speed while maintaining the simplicity of the coding.

Figure 2 shows the file organization of SimCell version 0.8. For the two

```
   811 define.h
   298 main.cc
  1944 spe.cc
  1437 spuinst.cc
   722 eib.cc
   171 memory.cc
   220 loaderbe.cc
------------------------
  5904 total
```

Fig. 2 The file organization of SimCell version 0.8. The first column is the number of lines. The second column is the file name. The total number of lines is 5,904. This code size is small for a multi-core processor simulator.

```
 1 int main(int argc, char **argv)
 2 {
 3   Chip *chip = new Chip();
 4   Spe *spe[MAX_SPE_NUM];
 5   new MainMemory(chip);
 6   new Mmu(chip);
 7   Eib *eib(chip);
 8
 9   for (int i = 0; i < MAX_SPE_NUM; i++)
10     spe[i] = new Spe(chip, i + 1);
11   eib = new Eib(chip);
12
13   initialize(chip, argv);
14
15   while (loopcond(chip) == LOOP_RUN) {
16     chip->cycle++;
17     for (int i = 0; i < chip->spe_num; i++) {
18       spe[i]->spu->step();
19       spe[i]->mfc->step();
20     }
21     eib->step();
22   }
23   finalize(chip);
24   return 0;
25 }
```

Fig. 3 The main function of SimCell version 0.8. A small part is simplified for ease of understanding.

requirements we use C++ as a description language. The line count of the source code, including comments and empty lines, is shown to the left of the file name. In SimCell version 0.8, among 194 instructions of SPE, 192 instructions are implemented excluding two unnecessary instructions (iret, bisled). The code of Spe class is described in spe.cc, spuinst.cc and define.h. These codes are the main part and they have approximately 4,200 lines. A total code of 5,904 lines is a very small as a simulator of a multi-core processor. We gave priority to the simplicity and the readability of the code. In order to maintain a high readability, we do not use global variables. In addition, we do not use the goto statement, which would complicate the control structure.

Figure 3 shows the main function of SimCell version 0.8. A small part of the code has been simplified for ease of understanding. The main loop is from the 15th line to the 22nd line. One iteration of the main loop corresponds to 1 clock cycle of the hardware. The *step* method of all modules is called in one iteration (18th line and 19th line) and is processed. Regardless of the running state of a module, the method is always called. Thus, the simulation speed is slow. However, there is an advantage in that the description becomes concise and the readability improves. Other modules are described concisely, similar to the main function. Thus, we are keeping a small amount of code, a simple structure, and a high readability for SimCell.

SimCell is portable and works in various environments. We confirm that SimCell version 0.8 works on Linux with Intel architecture or PowerPC architecture, and Cygwin. Since we do not use uncommon libraries, we can compile the code with general C++ compilers.

3.3 PPE is not Implemented

SimCell version 0.8 works as a simulator of the homogeneous chip multiprocessor, in which several SPEs and the main memory were connected by a network. PPE is not implemented on this version of SimCell. It causes some programming limitation. However, we can execute various parallel applications by only an SPE program with an appropriate pre-edit.

We adopt a programming model in which the SPE performs almost all of the tasks in the application, and the PPE handles only the initial process, such as the data preparation. In the simulator, the image file of the main memory is read when the initial process has finished, and the simulation is started from a process of the SPE. The process of the SPE reads data from the mailbox and starts the simulation. The mailbox transfers 4 bytes of data. If the SPE cannot read the data from the mailbox, its process stops until the data arrives. The SPE starts the process when it has been notified of the first address of the data from the PPE by using the mailbox.

In the memory image file, not only the contents of the main memory but also the mapping address of the Local Store and the contents of the mailbox are described in text format. Using the memory image file, we can start a simulation at the point at which the SPE obtains data from the mailbox.

Let us think about the making of the memory image. With a Cell/B.E. machine, the memory image file can be made in the PPE program easily by adding a function that outputs the contents of the main memory. However, if the memory

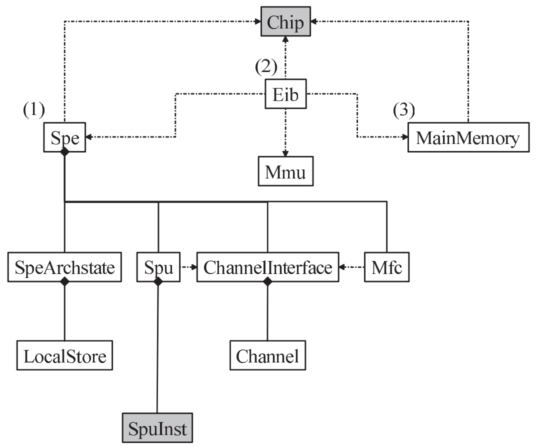

Fig. 4 The software architecture of SimCell. This diagram shows the dependency relations and the reference of each class.

image file for the simulation treats more SPEs than the number of SPEs of the actual machine, obtaining the mapping address of the Local Store will be difficult. Such a memory image file must be described manually, but the description of this part of the SPE comprises a few lines and is easy to write. Without a Cell/B.E. machine, the memory image file can also be made on a machine such as an Intel processor.

To run a simulation, we need a binary of SPE ISA. SimCell interprets the binary generated by an actual machine or by a cross compiler and loads it to the module of the Local Store.

Toshiba released a processor equipped with SPEs called SpursEngine. The configuration of this processor is similar to the model of SimCell version 0.8. As such, parallel computation by SPEs without PPE is practical.

3.4 Implementation Issues

The structure of the source code is understood easily because we implemented the hardware unit of Cell/B.E. as a class.

Figure 4 shows the relation between the classes in SimCell. This figure expresses the relation whereby the object of the different class. A solid line that ties two classes means that the class with a diamond mark generates the other class. For example, the object of the Spu class or the Mfc class is generated in the Spe class. Two classes connected by a dotted arrow indicate that the class at the tail of the arrow refers to the class at the head of the arrow. Note that we illustrate only the reference from central classes, because most classes refer to the Chip class.

In Fig. 4, the white classes are those for hardware units. The gray classes, on the other hand, require attention. These classes are added in order to improve the readability of the source code, and they do not implement the hardware in Cell/B.E. processor. For example, the Chip class stores information about the simulator configuration, such as the number of working SPEs and the communication latency.

We describe three key classes, denoted (1) through (3) in Fig. 4.

(1) Spe class This class implements the function of the SPE, and generates the object of the SpeArchstate class, the Spu class, the LocalStore class, the ChannelInterface class, and the Mfc class. The SpeArchstate class holds the value of a register and the program counter of the SPE. The LocalStore class corresponds to Local Store of the SPE. The Mfc class corresponds to the MFC of the SPE. The ChannelInterface class corresponds to the channel interface of the SPE. This class generates several objects of the Channel class. The Mfc class and the Spu class refer to the ChannelInterface class and supply a function of DMA transfer. The Spu class generates the object of the SpuInst class. The SpuInst class does not implement the hardware unit. It holds the information about an instruction being executed in order to improve the readability of the source code.

(2) Eib class This class handles the communication by arbitrating the requests of all units. For example, when a certain SPE performs a DMA transfer, this class schedules the communication and forwards the data with an appropriate timing. It is possible to set the parameters regarding the latency of the communication between units, the latency of the memory access, and the requirement for the maximum number of accesses to the main memory. Thus, the simulation of multi-core for various communication overheads is done easily. The Eib class refers to the Mmu class for address translation. In the actual Cell/B.E., each unit has its own MMU that translates addresses.

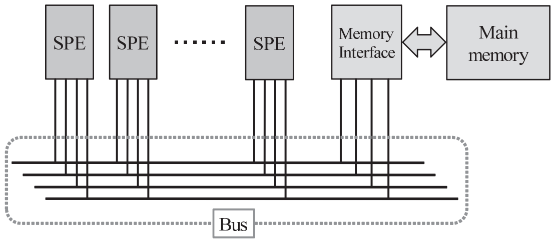

Fig. 5 The network architecture of SimCell.

In the SimCell, the Eib class refers to the Mmu class that implements one central MMU unit. The Mmu class manages the information of the Local Store that is mapped to the logical address space.

(3) MainMemory class This class is for the main memory and implements a 32 bit virtual memory. The address space is divided and managed by the page. The page is allocated to a real memory and referred. The results of the simulation are stored in this class.

Figure 5 shows the network architecture of SimCell. All SPEs and the memory interface are connected to buses. The network model is different from the four ring buses of Cell/B.E. processor. Units are connected to a simple bus rather than a ring bus. The arbitration of the bus is such that the following communication waits until the preceding communication finishes. The number of this bus can be set by a parameter.

SimCell separates the communication latency into the communication between cores and the communication with the main memory. The communication latency between cores is set based on the sending and receiving of overheads and the bandwidth of the bus. The communication latency with the main memory is set based on the overheads and the bandwidth of the memory access, as well as the overheads of the communication between cores. These overheads and the bandwidth can also be set by a parameter.

3.5 Simulation Accuracy and Speed

Figures 6 and **7** show the latency of DMA transfer. The latency in Fig. 6 is measured on PLAYSTATION 3, it implements the Cell/B.E. processor. The latency in Fig. 7 is measured on SimCell. The value is the average of repeated DMA transfer latency for 4 bytes to 16 K bytes of message size. This result is similar to the result in Ref. 2).

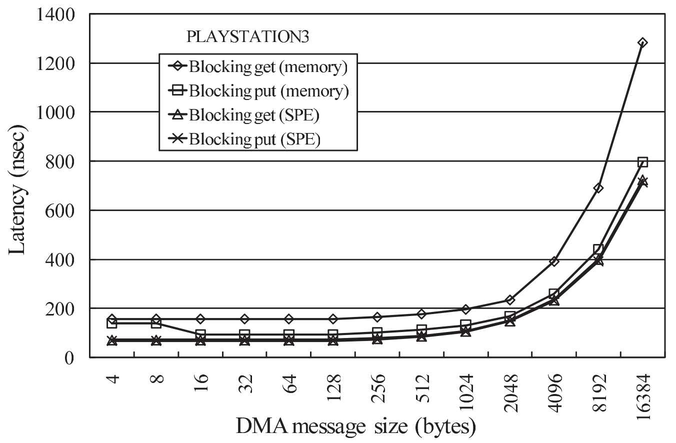

Fig. 6 The communication latency measured on PLAYSTATION 3 with Cell/B.E. processor.

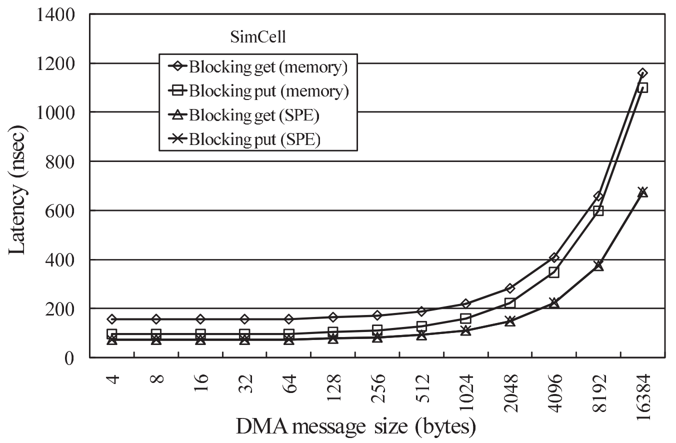

Fig. 7 The communication latency measured on SimCell. The network parameter is set from the measurement result of the real system.

The Blocking get (memory) means that the SPE reads data from the main memory. The Blocking put (memory) means that the SPE writes data to the main memory. The Blocking get (SPE) and the Blocking put (SPE) means the communication between two SPEs.

These graphs show that the communication latency on SimCell is close to the latency on PLAYSTATION 3 especially in the small range of a message size. In the large range of a message size, there is a gap of latency between SimCell and PLAYSTATION 3. But the increasing tendency of latency on SimCell corresponds to the tendency on PLAYSTATION 3. Therefore SimCell enables to simulate the performance to some extent.

Then we show the simulation speed of SimCell. To implement SimCell, we gave priority to the simplicity and the readability of the source code, while maintaining its practical simulation speed. So the simulation speed is not so fast but not too slow either. The computer used for simulation is described below:

- Intel Xeon (2.8 GHz, 512 Kbytes L2 cache)
- 1 GB Main memory
- Fedora core 6 (Linux kernel 2.6.20)
- ICC (Intel C compiler) Version 10.0, GCC Version 4.1.1

The benchmark program used herein is based on the quick sort. We explain the behavior of this benchmark. First, each SPE reads 512 bytes of data by DMA transfer from different regions of the main memory. Next, each SPE sorts the 512 bytes of data and writes them to the main memory by DMA transfer. Each SPE repeats these processes until all tasks have been completed. We prepare 4 Mbytes of initial data in the main memory. One task consists of 512 bytes of data.

The times of DMA transfers that each SPE does changes with the number of used SPEs. Since there is no data dependency in this application, we can execute each task completely in parallel. Therefore, it is convenient to measure the simulation speed when simulating from a few to hundreds of SPEs. To generate SimCell binary, we use gcc version 4.1.1 with optimization options O0, O2, and O3 and Intel compiler version 10.0 with optimization option O3.

Figure 8 shows the simulation speeds obtained using different compilers. The x-axis is the number of SPE, and the y-axis is the simulation speed. The unit of the y-axis is MIPS (Million Simulated Instructions per Second), which are obtained by the summation of all SPE's simulated instructions per second.

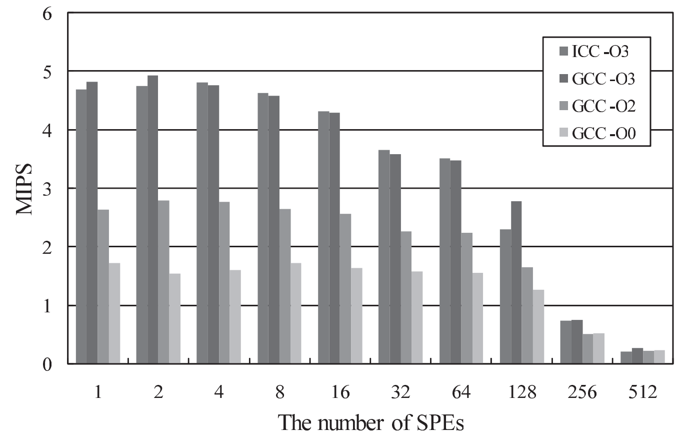

Fig. 8 The simulation speed of SimCell compiled by different compilers and optimization flags. The y-axis is the number of total simulated instructions per second (MIPS).

Table 1 The comparison of the execution/simulation time between a real computer (PlayStation3) and SimCell.

	PS3	SimCell	slowdown
1 SPE	223 [msec]	39.6 [sec]	177.6
2 SPE	112 [msec]	38.6 [sec]	344.6
4 SPE	56 [msec]	38.1 [sec]	680.4

The simulation speed does not simply decrease if we increase the number of SPEs. In the case of 512 SPEs, the MIPS value is approximately 0.2, and the simulation speed is slow. On the other hand, when the number of SPEs is approximately 16, we achieve a practical performance of about 4 MIPS using the Intel compiler or gcc with the optimization option O3. For approximately the same number of SPEs, in the case of gcc with optimization option O2, the performance is about 3 MIPS, and in the case of gcc without the optimization option, the performance is about 1 MIPS.

Table 1 summarizes the execution time measured on an actual machine (PLAYSTATION 3 or PS3) of Cell/B.E. and on SimCell for the same application. We compiled SimCell using the Intel compiler version 10.0 with optimization option O3. Simce the number of available SPEs on PS3 is six, we compare the execution times when implementing one, two, and four SPEs. Note that the unit of execution time on SimCell is a second, whereas that on PS3 is a millisecond.

From Table 1, we see that SimCell requires an execution time 100 to 1,000 times greater than that of an actual machine. Since the process of each SPE is made sequentially, the execution time of SimCell does not change significantly.

When using SimCell, the simulation slowdown is approximately 100 to 1,000. However, the simulation speed is still practical for certain applications. Nevertheless, the adaptation of a speedup technique remains a consideration.

3.6 Visualization Tool LSView

In the programming of Cell/B.E., an application does not work normally if the timing of DMA transfers and the alignments of data are not described correctly. For programmers, it is useful to know the behavior of the Local Store in order to avoid such a basic failure.

We supply a tool for visualizing the Local Store called LSView. We developed LSView using Java. To use LSView, an application must first be executed by SimCell and a log of the Local Store must be generated. LSView reads the log file and displays the state of the reference of the Local Store as an animated image. The log is a record of the access frequency of the Local Store during the specified period set by the user. LSView generates a bitmap image, which assigns a region of 4 bytes to each pixel. Since the capacity of the Local Store is 256 Kbytes, the image is 256×256 pixels. We can confirm the state of the Local Store at each time as an animated image by displaying images continually.

Figure 9 shows a screenshot at step 0 (initial stage of application execution) in which the behavior of four Local Stores is visualized using LSView. Each SPE executes the application using 32 KB of data received by DMA transfer from different regions of the main memory. The data is sorted and then written back to the main memory. This is similar to the application in Section 3.5.

Figure 9 indicates that the behavior of the Local Store is different in each SPE. The color of LSView becomes darker if the reference frequency is high. Instructions of the application are stored at the top of the square region (an upper address of the Local Store), and this part is referred to frequently. The quantity of data to be sorted is 32 Kbytes, so 1/4 of the region of the Local Store is used. As the sorting process progresses, the color of data region becomes

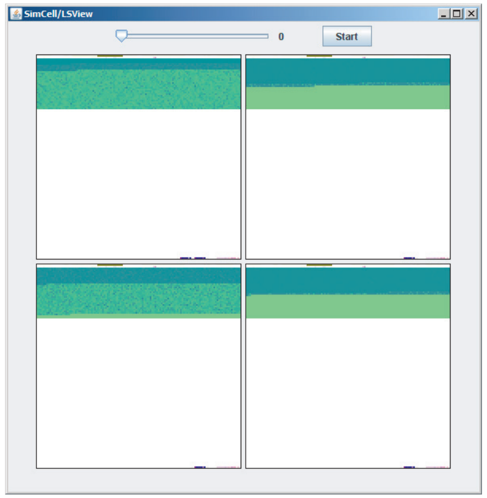

Fig. 9 The screenshot of the LSView at step 0. The memory access behavior of four Local Stores are shown.

darker. The bottom of the square region (a lower address of the Local Store) is used as a stack area. The colors of the region in which instructions are stored, the region to which the load/store instruction refers, and the region to which the DMA transfer refers are different.

Figure 10 shows a screenshot at step 5 of LSView. The color becomes darker for all Local Stores as the program proceeds. The bottom of the square region for the stack area is enlarging by the function calls.

Debugging of the parallel application is difficult and the support tools, such as conventional debuggers, become important. As described in this section, we can understand the behavior of the application intuitively by visualizing the state of the memory reference continually using SimCell and LSView. In addition, we can significantly reduce the load of the programmer.

3.7 Brief Development History

The development of SimCell began in October 2006, and we made (to a certain extent) the first simulator of the functional level of the SPE in February 2007. By that time, half of the instructions of the SPE were implemented.

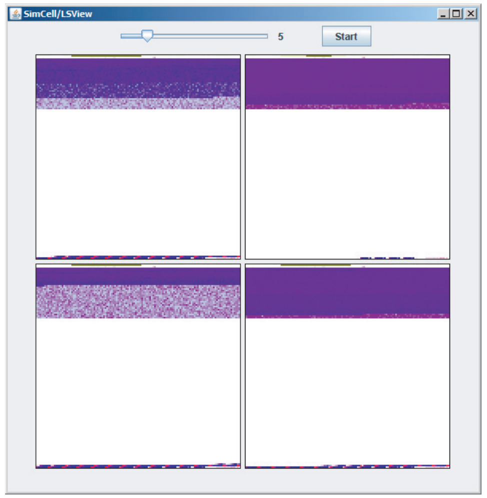

Fig. 10 The screenshot of the LSView at step 5. The memory access behavior of four Local Stores are shown.

In March 2007, a simple network module was implemented, and a simple multi-core simulation was done. This version was the prototype of the current SimCell.

From April 2007, we improved the functional level simulator of the SPE and implemented almost all of the instructions of the SPE by October 2007. Afterwards, we continued to improve the network module and released SimCell version 0.8 in March 2008.

4. Real Case Study Using SimCell

As an example using SimCell, we examine the feasibility of Cache-Core. Cache-Core is one of the software cache[17]. In Cache-Core, a SPE is used only as a software-controlled data cache. We show that the evaluation with a modified hardware organization can be performed easily by simply changing the parameters of SimCell, without modifying the source code.

4.1 Cache-Core Mechanism

In an application using SPEs, it is difficult for programmers to schedule the memory access. In such cases, there is a technique to improve the performance of the application. We expanded the software cache, and proposed a method called Cache-Core. In Cache-Core, a SPE is used as a software-controlled data cache. Here, we describe the performance of Cache-Core.

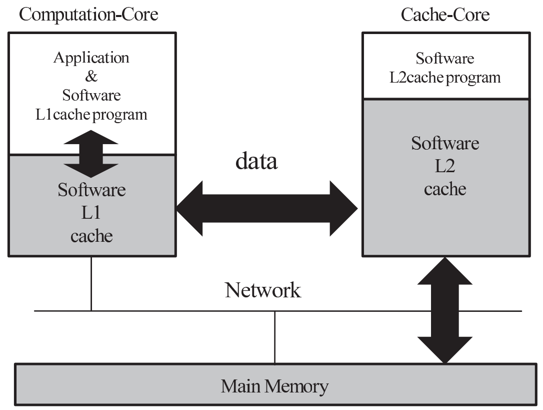

Fig. 11 The Cache-Core mechanism. Two SPEs are used. One is for an application program, and the other is for the L2 software cache program.

Figure 11 shows the mechanism of Cache-Core. We implement an L1 software data cache on the SPE executing an application (Computation-Core). In the SPE of Cache-Core, we implemented an L2 software data cache. We explain the behavior of the software cache using Cache-Core. When the Computation-Core misses the L1 data cache (on Computation-Core), it requires data from L2 data cache (on Cache-Core). On-chip communication is faster than the communication outside the chip, such as memory access. Therefore, we can anticipate the performance enhancement of the application by using the Local Store on Cache-Core as an L2 cache.

We next evaluate the performance of Cache-Core with SimCell. In this evaluation, we do not simulate each core at the cycle level. Instead, we examine the influence of the overhead in communication between cores. Using SimCell version 0.8, we can evaluate the overhead while changing the communication latency.

Here, the change in the communication latency is the total number of transfer cycles when a packet (128 Bytes) in Cell/B.E. arrives at the unit from another unit. That is to say, the communication latency is the total number of cycles

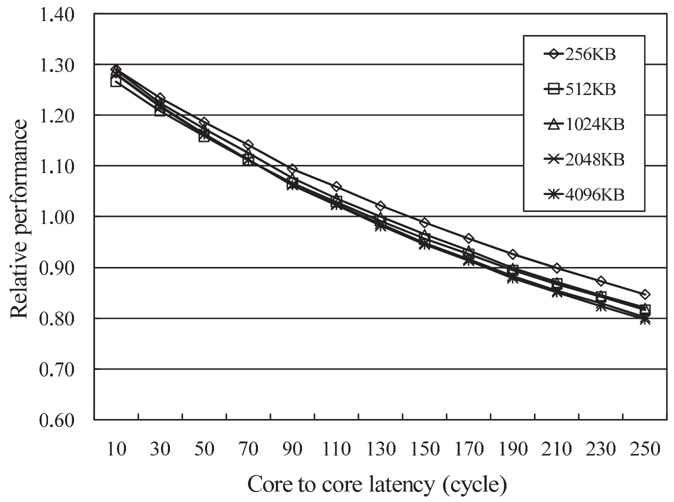

Fig. 12 The performance of Cache-Core while changing the core-to-core communication latency. The y-axis is the relative performance normalized by the performance of a single core without an L2 Cache-Core.

before the communication completes after the SPE has issued a DMA transfer command to forward 128 bytes of data. It is difficult to change the communication latency in an actual machine. However, we can change the communication latency using SimCell.

Figure 12 shows the effect of Cache-Core when we change the communication latency between cores. We use a quick sort as a benchmark and set the capacity of the L1 cache as 2 Kbytes and that of the L2 cache as 64 Kbytes. It shows the result of from 256 Kbytes to 4,096 Kbytes sort size. The x-axis is the latency of the communication on the chip, and the y-axis is the relative performance. We normalized the performance with Cache-Core by the configuration without Cache-Core.

Figure 13 is a screen shot of Cache-Core visualized using LSView. Computation-Core corresponds to the left area, and Cache-Core corresponds to the right area. We can confirm the behavior of DMA transfer on Cache-Core in an animation.

The evaluation with SimCell reveals that the performance enhancement by

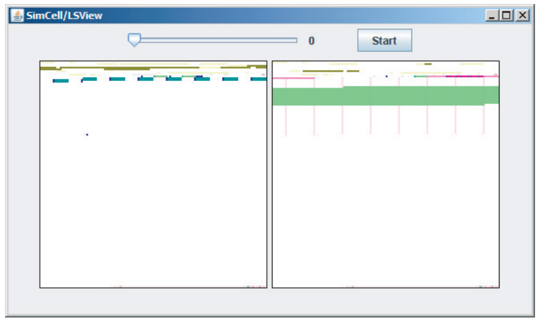

Fig. 13 The screenshot of the LSView to visualize the Local Store usage. The memory access behaviors of two SPE Local Stores are shown.

Cache-Core is realized in the realistic range in which the communication latency is lower than 130 cycles. In this way, we can easily evaluate the performance of the modified hardware simply by changing the parameters of SimCell, without modifying the source code.

5. Advanced Case Study

5.1 SimCell/CA

In previous sections, we described the functional level simulator SimCell version 0.8. Here, a functional level simulator is the simulator of a simplified processor model that executes one instruction in a single cycle. The functional level simulator is used in various cases. However, the difference between the actual machine and the functional simulator in the execution cycle becomes large. Therefore, a high accuracy comparison in a sensitive evaluation, such as the evaluation of pipeline configuration influences, becomes difficult. Based on this background, we extended SimCell version 0.8 to SimCell/CA in order to simulate the SPE at the cycle level. The functional level simulator, SimCell, has a high extensibility, so we were able to build a cycle level simulator by adding only approximately 160 lines to the source code.

We examined the simulation accuracy of SimCell/CA by comparing the number of execution cycles on SimCell/CA, PLAYSTATION 3, and the IBM simulator using an application for 1 SPE.

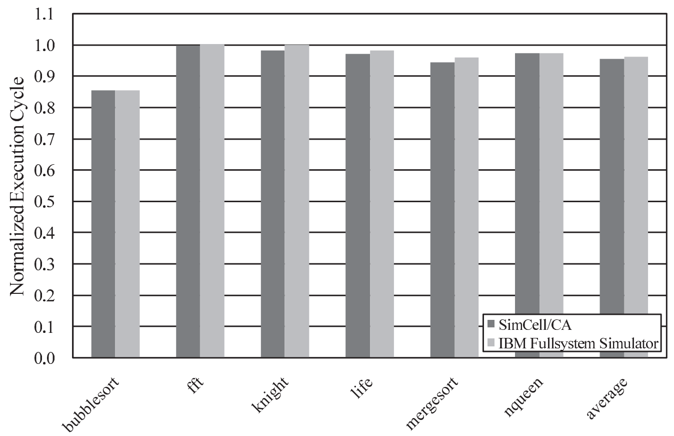

Fig. 14 The accuracy of SimCell/CA. The gap between SimCell/CA and the IBM full-system simulator is small.

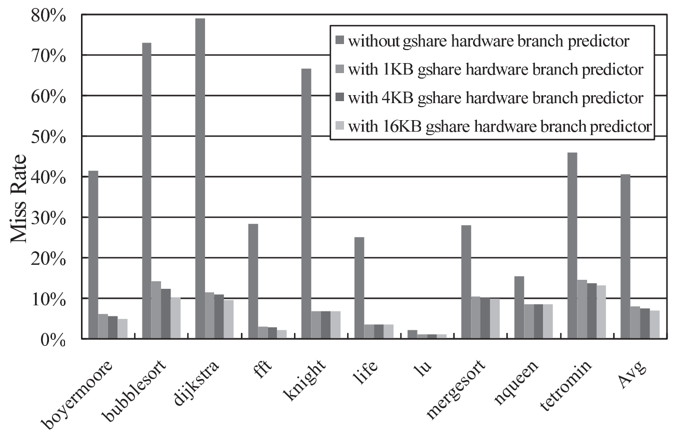

Fig. 15 The comparison of misprediction rate with/without gshare hardware branch prediction.

In **Fig. 14**, we present the results of the comparison of an execution cycle on SimCell/CA and the IBM Full-System Simulator for the Cell Broadband Engine Processor. Here, we normalize the execution cycle to the execution on the PLAYSTATION 3. We used six applications, namely, bubblesort, fft, knight (problem of the knight itinerancy), life (a board game), mergesort and nqueen. None of the applications have DMA transfer.

Some of the results for the execution cycle are short on SimCell/CA compared to an actual machine. However, the gap between SimCell/CA and IBM simulator is as large as 1.8%, and is 0.8% on average. Thus, SimCell/CA achieves a good simulation accuracy.

5.2 Case Study of Branch Prediction on Cell/B.E.

The SPE of Cell/B.E. does not have a hardware branch predictor. Applications are basically based on software branch prediction.

However, the pipeline of the SPE is long, and the branch miss-penalty has a significant influence on the execution performance of the application. As an example using SimCell/CA, we show the performance of an SPE with a hardware branch predictor. Moreover, we clarify that we can consider the performance changing the hardware organization, which cannot be changed easily on the actual machine, by using SimCell/CA.

We assume the following as a branch predictor. A branch prediction is made in a cycle. The prediction is reflected immediately, and the following instruction of the branch instruction is fetched without a break. The penalty when branch prediction misses is 17–18 cycles. (This is the same penalty in the actual Cell/B.E.) We used gshare [18] for branch prediction. The size of PHT (Pattern History Table) is 1 K bytes to 16 K bytes.

Applications for evaluation are boyermoore (text search), bubblesort, dijkstra (shortest path problem), fft, knight (problem of the knight itinerancy), life (a board game), lu (LU decomposition), mergesort, nqueen, and tetromin (puzzle of tetrominoe). We used spu-gcc version 4.1.1 as a compiler, and the optimization option is O2. We used the binary file, from which hint branch instructions were removed, for simulation with the hardware branch predictor.

Figure 15 shows the misprediction rate for the cases with and without branch prediction. In addition, we show the performance with a branch predictor in **Fig. 16**. Here, we normalize the performance of the SPE without a branch predictor. The misprediction rate is improved significantly (from 40% to 7% on average). Furthermore, the maximum improvement in performance is 81%, and

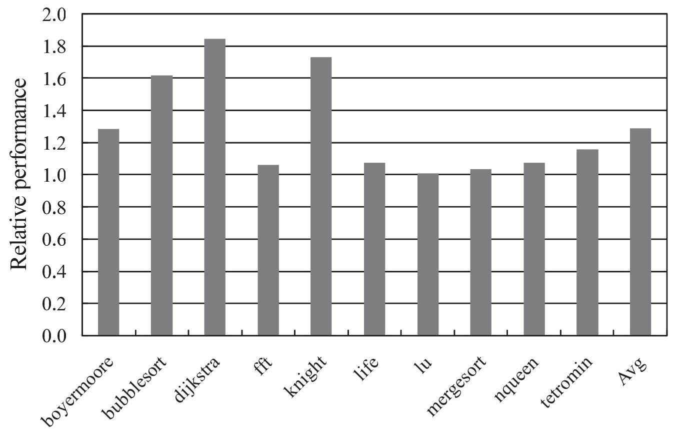

Fig. 16 The speedup by gshare hardware branch prediction. The base performance is the configuration without gshare branch prediction.

the average improvement in performance is 28%.

This example of branch prediction indicates that an improvement in performance can be expected. However, we can obtain detailed quantitative data, such as the misprediction rate or the performance enhancement rate, by using SimCell/CA.

6. Conclusions

For the research on computer architectures, processor simulators are fundamental software. We developed a new open source multi-core processor simulator SimCell from scratch. SimCell is modeled around the Cell Broadband Engine.

We described the advantages of the functional level simulator SimCell. From the verification of the simulation accuracy and the speed, we confirmed that SimCell is available to evaluate the performance to some extent and achieves a practical simulation speed. Through a real case study using SimCell, we clarify the usefulness of SimCell for processor architecture research.

We developed a visualization tool called LSView. LSView reads a log file of SimCell and visualizes the behavior of Local Store in an animation. When debugging a parallel application, a visualization tool such as LSView reduces the load of programmers.

As a cycle-accurate version of SimCell, we developed SimCell/CA. It was built by adding approximately 160 lines to SimCell. To demonstrate the accuracy of SimCell/CA, we compared the execution cycles on SimCell/CA and IBM simulator. The gap of execution cycles is 0.8% on average. Thus, SimCell/CA achieves a good simulation accuracy. Through an advanced case study using SimCell/CA, we clarify the usefulness of SimCell/CA for processor architecture research.

We released SimCell version 0.8 as an open source project in March 2008. The URL of SimCell Web page is http://www.arch.cs.titech.ac.jp/SimCell/.

Acknowledgments This work is supported in part by Grants-in-Aid for Young Scientists (B) No.18700042. We would like to thank Yutaka Sasaki and Shinya Tahara. Yutaka developed the first functional level simulator of SPE. Shinya implemented most of SPE instructions on SimCell.

References

1) Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R. and Shippy, D.: Introduction to the Cell multiprocessor, *IBM Journal of Research and Development*, Vol.49, No.4/5, pp.589–604 (2005).
2) Kistler, M., Perrone, M. and Petrini, F.: Cell Multiprocessor Communication Network: Built for Speed, *IEEE Micro*, Vol.26, No.3, pp.10–23 (2006).
3) Gschwind, M., Hofstee, H.P., Flachs, B., Hopkins, M., Watanabe, Y. and Yamazaki, T.: Synergistic Processing in Cell's Multicore Architecture, *IEEE Micro*, Vol.26, No.2, pp.10–24 (2006).
4) Moriya, A., Fujieda, N., Sato, S. and Kise, K.: The Multi-Function Cache Core Architecture to Enhance the Memory Performance on Many-Core processors, *SACSIS 2008*, pp.421–430 (2008).
5) Bohrer, P., Elnozahy, M., Gheith, A., Lefurgy, C., Nakra, T., Peterson, J., Rajamony, R., Rockhold, R., Shafi, H. and Simpson, R.: Mambo . A Full System Simulator for the PowerPC Architecture, *ACM SIGMETRICS Performance Evaluation Review*, Vol.31, No.4, pp.8–12 (2004).
6) Fujieda, N., Sato, S. and Kise, K.: The Examination of Software Branch Predictions Considering Dual Branch Hints, *IPSJ SIG Technical Report 2008-ARC-177*, pp.121–126 (2008).
7) Sato, S., Fujieda, N., Tahara, S. and Kise, K.: Design and Implementation of Cell BE Functional Simulator, *IPSJ SIG Technical Report 2007-ARC-174*, pp.187–192 (2007).
8) Sato, S., Fujieda, N., Tahara, S. and Kise, K.: The Cell BE Functional Simulator

SimCell that is Designed and Implemented Pursuing Practical and Simply Coded Simulator, *ComSys2007*, pp.39–47 (2007).
9) Sato, S., Fujieda, N., Moriya, A. and Kise, K.: Processor Simulator SimCell to Accelerate Research on Many-core Processor Architectures, *Proc. 2008 Workshop on Cell Systems and Applications*, pp.119–127 (2008).
10) RAMP: Research Accelerator for Multiple Processors, University of California at Berkeley, http://ramp.eecs.berkeley.edu/.
11) Burger, D. and Austin, T.M.: The SimpleScalar Tool Set, Version 2.0, *University of Wisconsin-Madison Computer Sciences Department Techical Report*, No.1342 (1997).
12) Binkert, N.L., Dreslinski, R.G., Hsu, L.R., Lim, K.T., Saidi, A.G. and Reinhardt, S.K.: The M5 Simulator: Modeling Networked Systems, *IEEE Micro*, Vol.26, pp.52–60 (2006).
13) Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A. and Werner, B.: Simics: A Full System Simulation Platform, *IEEE Computer*, Vol.35, No.2, pp.50–58 (2002).
14) UNISIM: UNIted SIMulation environment. http://unisim.org/site/.
15) Cabarcas, F., Rico, A., Rodenas, D., Martorell, X., Ramirez, A. and Ayguade, E.: CellSim: A Validated Modular Heterogeneous Multiprocessor Simulator, *XVII Jornadas de paralelismo, JP 2007* (2007).
16) Kise, K., Katagiri, T., Honda, H. and Yuba, T.: The SimCore/Alpha Functional Simulator, *Workshop on Computer Architecture Education*, pp.128–135 (2004).
17) Balart, J., Gonzalez, M., Martorell, X., Ayguade, D., Sura, Z., Chen, T., Zhang, T., O'brien, K. and O'brien, K.: A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor, *LCPC 2007: 20th International Workshop on Languages and Compilers for Parallel Computing* (2007).
18) McFarling, S.: Combining Branch Predictors, *WRL Technical Note TN-36* (1993).

(Received July 23, 2008)
(Accepted October 7, 2008)

Shimpei Sato received the B.E. degree from Tokyo Institute of Technology in 2007. He is currently a Master Course student of the Graduate School of Informarion Science and Engineering, Tokyo Institute of Technology. His research interests include Many-Core processor architecture and Network-on-Chip.

Naoki Fujieda received the B.E. degree from Tokyo Institute of Technology in 2008. He is currently a Master Course student of the Graduate School of Informarion Science and Engineering, Tokyo Institute of Technology. His research interests include processor architecture.

Akira Moriya received the B.E. degree from Tokyo Institute of Technology in 2007. He is currently a Master Course student of the Graduate School of Information Science and Engineering, Tokyo Institute of Technology. His research intersts include computer architecture and embedded system.

Kenji Kise received the B.E. degree from Nagoya University in 1995, the M.E. degree and the Ph.D. degree in information engineering from the University of Tokyo in 1997 and 2000 respectively. He is currently an Assistant Professor of the Graduate School of Information Science and Engineering, Tokyo Institute of Technology. His research interests include computer architecture and parallel processing.