IPSJ Transactions on System LSI Design Methodology Vol. 3 179–193 (Aug. 2010)

Regular Paper

Efficient Design Space Exploration at System Level with Automatic Profiler Instrumentation

Seiya Shibata,[†1,†2] Yuki Ando,[†1] Shinya Honda,[†1]
Hiroyuki Tomiyama[†3] and Hiroaki Takada[†1]

As the complexity of embedded systems grows, design space exploration at a system level plays a more important role than before. In the system-level design, system designers start from describing functionalities of the system as processes and channels, and then decide mapping of them to various Processing Elements (PEs) including processors and dedicated hardware modules. A mapping decision is evaluated by simulation or FPGA-based prototyping. Designers iterate mapping and evaluation until all design requirements are met. We have developed two profilers, a process profiler and a memory profiler, for FPGA-based performance analysis of design candidates. The process profiler records a trace of process activations, while the memory profiler records a trace of channel accesses. According to mapping of processes to PEs, the profilers are automatically configured and instrumented into FPGA-based system prototypes by a system-level design tool that we have developed. Designers therefore need to manually modify neither the system description nor the profilers upon each change of process mapping. In order to demonstrate the effectiveness of our profilers, two case studies are conducted where the profiles are used for design space exploration of AES encryption and MPEG4 decoding systems.

1. Introduction

In order to design embedded systems of high quality in a short time, fast and accurate profiling and evaluation are musts for design space exploration. As the complexity of embedded systems grows to the extent of MPSoCs (multiprocessor system on a chip), design space exploration at a system level plays a more important role than before. In the system-level design, system designers start from describing functionalities of the system as processes and channels which indicate computations and communications among processes, respectively. Then the designers decide mapping of them to various Processing Elements (PEs) including processors and dedicated hardware modules [1]. A mapping decision is evaluated by simulation or FPGA-based prototyping. The designers iterate mapping and evaluation until all design requirements are met.

Performance evaluation of mapping decisions requires timed descriptions. Recent system-level design tools provide automatic synthesis capabilities of timed descriptions from untimed descriptions and mapping [2)–4)]. These tools convert processes which are mapped on software into compilable program modules and processes which are mapped on hardware into synthesizable RTL circuits. Channels are converted into appropriate communication modules. The synthesized timed descriptions can be evaluated by simulation or FPGA-based prototyping.

A number of researches on simulation-based evaluation were conducted in the past. Aiming at functional verification of hardware-software systems, fast but inaccurate simulation techniques were proposed [5),6)]. In contrast, cycle-accurate hardware simulation tools [7),8)] were widely accepted for accurate but slow evaluation in an industrial domain. Fummi, et al. proposed a cosimulation framework for both verification and evaluation [9)]. Their framework provides synchronization mechanisms, which vary in speed and accuracy, on communications between hardware and software. Designers therefore can use an appropriate mechanism depending on their needs (verification or evaluation). However, since speed and accuracy are incompatible with each other on simulations on a host PC, simulations are often inappropriate for design space exploration, especially for recent complex systems.

Another approach to performance evaluation is FPGA-based prototyping. FPGA-based prototypes achieve both high accuracy and speed, and are appropriate for iteration of mapping and evaluation. One disadvantage of FPGA-based prototypes is that internal states of the system are unobservable without additional modification for system descriptions. Since recent systems have complex dependencies and concurrency among processes, profiling capabilities are essential to find out bottlenecks and to help designers decide mapping alternatives. However, manual modification for profiling is time-consuming and error-prone. In order to prune design candidates efficiently and find the best choice quickly, automatic instrumentation for profiling is necessary.

†1 Graduate School of Information Science, Nagoya University
†2 Japan Society for the Promotion of Science
†3 College of Science and Engineering, Ritsumeikan University

We have developed two profilers, a process profiler and a memory profiler, for FPGA-based performance analysis of design candidates. The process profiler records a trace of process activations, while the memory profiler records a trace of memory channel accesses. In our framework, systems are described at a high level and FPGA-based system prototypes are automatically synthesized by our system-level design tool, named SystemBuilder. According to mapping of processes to PEs, the profilers are automatically configured and instrumented into the FPGA-based system prototypes by SystemBuilder. Designers therefore need to manually modify neither the system description nor the profilers upon each change of process mapping. The profilers allow fast and accurate performance evaluation of the systems which have complex dependency and concurrency with the profilers using an FPGA. In summary, major contributions of our profilers on design space exploration are

- automatic instrumentation of the profilers with support of a system-level design tool,
- fast and accurate FPGA-based evaluation and profiling,
- and profiling capabilities for concurrent MPSoCs.

The rest of this paper is organized as follows. First, Section 2 presents a brief overview of related works about system evaluation techniques for design space exploration. Next, Section 3 explains our system-level design tool. Section 4 describes two proposed profilers and Section 5 shows the effectiveness of the profilers through two case studies. Finally Section 6 concludes this paper with a summary.

2. Related Works

There are many approaches which provide efficient evaluation environments for design space exploration.

ARTS [10] and TAPES [11] are system-level simulation frameworks. ARTS is a framework for modeling and simulating MPSoCs. Given profiles of tasks to be executed on processing elements, ARTS simulates communications among tasks and calculates performance. TAPES provides a retargetable simulation framework with a given profile of the system functionality. These frameworks assume that profiles of the system at a system level are given prior to their simulation, therefore the accuracy of their simulation depends on the accuracy of profiles.

As for FPGA-based approaches to system-level design, automatic synthesis tools [2),4),12)] enabled designers to implement FPGA-based prototypes in a short time. By using IPs for debugging hardware provided by FPGA vendors (ChipScope [13], SignalTap [14]), designers can observe internal signals of an FPGA and analyze behavior of the system in detail. These approaches, however, need expertise of hardware and manual modification of the system (hence error-prone) for profiling. Valle, et al. proposed an environment on an FPGA for profiling software which is executed on multi-processor systems [15]. In their environment, clock inputs for the system under profiling can be controlled and the accuracy is guaranteed. Their environment, however, cannot handle dedicated hardware modules. Nunes, et al. proposed a profiler construction for multi-FPGA systems with high-level descriptions of systems [16]. Their approach has a similar concept with ours in assuming that systems are developed using specific communication channels between functionalities and the profilers are developed to cooperate with the channels. However, their profiler needs manual instrumentation.

In contrast with above tools, our profilers record traces of concurrent MPSoC systems, achieving both high accuracy and short execution time. System designers can profile the large number of design alternatives easily with automatic instrumentation of the profilers.

3. SystemBuilder

SystemBuilder is a system-level design toolkit developed in our prior work [3]. The main objective of SystemBuilder is to support design space exploration of embedded systems. In this work, we extended SystemBuilder to automatically instrument the systems with a process profiler and a memory profiler. In this section, we explain target architecture of SystemBuilder, a brief overview of the design flow achieved by SystemBuilder and the features of channels for later sections.

3.1 Target Architecture

SystemBuilder currently uses Altera's tools as its back-end for logic synthesis and place-and-route, and therefore, the target architecture of SystemBuilder

is restricted to one supported by Altera's tools. Specifically, Nios II soft-core processors with Avalon buses are supported by SystemBuilder at present.

There are no restrictions on the numbers of processors, hardware accelerator modules, memory modules and buses, as many as the FPGA device allows. The numbers of these modules and the interconnection between them are defined by designers in an input file of SystemBuilder (described as "architecture template" in Section 3.2).

Mapping of processes onto processors statically is determined at a design phase and is not changed at runtime. SystemBuilder also assumes that a single address space is shared by all the modules. The two assumptions are realistic and very popular in many embedded systems in order to meet real-time requirements [17].

It should be noted that, although SystemBuilder at present supports only Altera's FPGAs and their associated architectures, SystemBuilder can potentially support other devices and architectures[⋆1].

3.2 Design Flow

Figure 1 shows the design flow achieved with SystemBuilder. First, a system designer develops a "system-level description" to capture functionalities of the target system. The system-level description consists of "processes" and "channels". A process and a channel represent a computation component and interprocess communication at a high abstraction level, respectively. The processes may be mapped onto processors and hardware modules, and the channels onto buses, memories and other communication devices. The designer also specifies hardware architecture in an "architecture template" and mapping of the processes and the channels onto the hardware architecture in a "mapping specification". SystemBuilder automatically synthesizes descriptions of interconnections between processes (hereafter, this synthesis functionality is called as "communication synthesis"). The synthesized communication descriptions are generated in the C language and VHDL, depending on mapping of the processes and the channels. Next, SystemBuilder makes use of a compiler of the processors for software and a behavioral synthesis tool for hardware modules in order to obtain executable binaries and synthesizable RTL circuits, respectively. Processes mapped on software are compiled and linked with a Real-Time OS (RTOS). Finally, a configuration bitstream of the designed hardware architecture for an FPGA is synthesized by a logic synthesis tool from the RTL circuits and IPs of the processors and essential peripherals.

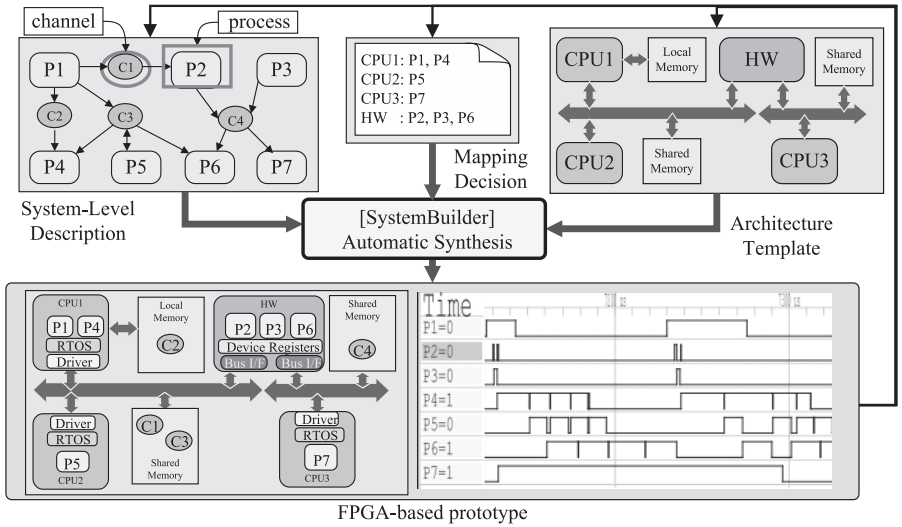


Fig. 1 Design flow of SystemBuilder.

After automatic synthesis of an FPGA-based prototype by SystemBuilder, the prototype is evaluated on an FPGA. If the prototype meets the designer's requirements, the design flow is completed. Otherwise, the designer may explore other implementations of the system by changing the system-level description, the architecture template and/or the mapping specification (denoted as feedback arrows from the FPGA-based prototype to inputs of SystemBuilder in Fig. 1). By iteration of changing inputs, automatic synthesis and evaluation, the system designer explore design space to find the best implementation of the system which meets his/her requirements. SystemBuilder realizes short turn-around-time of this iteration by providing the automatic synthesis capability.

In summary, there are following exploration opportunities during the design

⋆1 In actual, an earlier version of SystemBuilder supported Xilinx's architecture with Microblaze processors and the OPB bus [3].

flow with SystemBuilder:

- system-level description construction with processes and channels,
- hardware architectures consisting of processors, hardware modules, memories and buses,
- and mapping of processes and channels.

Note that SystemBuilder does not provide any automation technique to find the best implementation, and leaves to designers the decision on how to improve the system.

3.3 Features of The Channels

Here, we explain about the channels which we focused on the implementation of our profilers. SystemBuilder provides two types of channels: blocking channels and memory channels.

Blocking channels can be used for describing data/control dependencies between two processes. A receiver process of a blocking channel is forced to wait until a corresponding sender process writes data to the channel. A blocking channel is transformed to a FIFO hardware buffer or a queue API of an RTOS by communication synthesis depending on mapping of the processes. We implemented the process profiler utilizing these characteristics of blocking channels.

Memory channels represent storage of data transferred among processes. They are transformed to either of block memories on an FPGA or an off-chip memory, or an array of the C language. Memory channels mapped between software and hardware are implemented as an interface circuit of a hardware module in order to realize communication between hardware and software. In particular, the processes on hardware which access memory channels mapped onto off-chip memories are implemented to use interface circuits in order to access the memories. SystemBuilder generates a single interface circuit, which is shared by the processes, by default, and generates two or more interface circuits with the designers' specification. We implemented the memory profiler utilizing these characteristics of memory channels.

4. System-Level Profilers

We propose two profilers, a process profiler and a memory profiler. The profilers are automatically configured and instrumented into FPGA-based prototypes by SystemBuilder, and record traces of processes and memory accesses at runtime of the prototypes.

The process profiler has been developed to help designers analyze behaviors of processes taking concurrency and dependencies among processes into account. Since processes of recent embedded systems may have complex dependencies with each other, the mapping decision which maximizes the parallelism of processors and dedicated hardware modules is not obvious and needs to be explored. In order to find the optimal mapping of processes, evaluation of mapping decisions is not sufficient with only execution time of individual processes. It is important to record activation/wait timings of processes and analyze the parallel behavior of processes with the timings.

Processes have not only explicit dependencies among them represented by blocking channels but also implicit relationships which appear in accessing shared resources. The memory profiler has been developed to help designers analyze the effects of such conflicts at memory channels. Recent embedded systems which consist of concurrent processes often share memories for communication and resource reduction. Since simultaneous memory accesses for a single memory module cause conflicts and need to be handled sequentially by bus arbiters or memory interfaces with additional cycles, they may cause performance degradation. Therefore the memory profiler records traces of memory channels to analyze conflicts on shared memories.

It should be noted that the profilers do not restrict the capability of SystemBuilder. In other words, the profilers can be inserted into any system designed by SystemBuilder. In turn, since the current profilers are tightly coupled with SystemBuilder, the profilers can hardly be applied to system architectures which are not supported by SystemBuilder. Another restriction at present is that at most 16 processes can be profiled by the process profiler and at most 16 memory channels can be profiled by the memory profiler, but this restriction will be relaxed in near future.

In this section, we describe advantages, profiling flow and detail of the profilers.

4.1 Advantages

In order to explore design candidates efficiently, we developed the profilers which have following advantages:

(1) automatic instrumentation of the profilers at the system synthesis phase,
(2) common timeline between software and hardware,
(3) low profiling overhead on performance,
(4) visualization of traces for intuitive analysis.

Automatic instrumentation of the profilers is necessary for realizing smooth iteration of mapping and evaluation. Otherwise, designers have to manually modify system descriptions and the profilers upon each change of process mapping.

Common timeline between software and hardware is required since processes are mapped onto either software or hardware. In order to help designers find bottleneck processes out from both software and hardware, the profilers must record traces of them in a common timeline.

Performance overhead of the profilers must be small enough not to have influence on behaviors of processes. We especially gave priority to minimize performance degradation on development of the profilers. This is why we decided to implement major parts of the profilers in hardware. We paid less attention for area overhead than performance overhead since the profilers are removed from the final implementation of the system in our design flow.

The objective of visualization of traces is to help designers find bottlenecks out from complex process behavior.

4.2 Profiling Flow

Figure 2 shows the profiling flow of our profilers with SystemBuilder. The profiling flow starts from "system-level description" (denoted in Fig. 2), and SystemBuilder automatically generates an FPGA-based prototype according to a mapping decision of a designer. The prototype consists of "FPGA configuration" of hardware modules and "executable binary" of software, and the process profiler and the memory profiler are configured and instrumented in software and hardware automatically. Process trace and memory trace are recorded by executing the FPGA-based prototype. Then, "trace output" is transferred from the FPGA to the host PC. Finally, the trace output is transformed for analysis and visualization by "trace analyzer & visualizer".

After this flow, designers can analyze the prototype using the traces, and can make feedback for the inputs of SystemBuilder to find better system implementations.

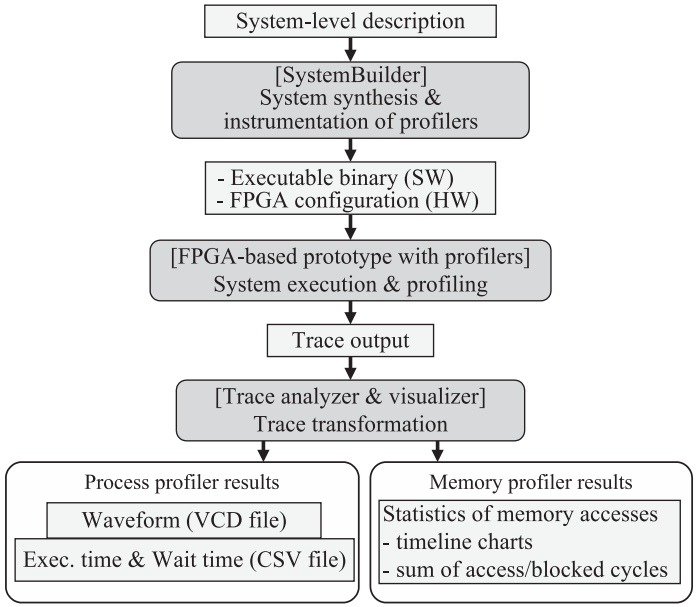


Fig. 2 Tool flow of the profilers supported by SystemBuilder.

Since capacities of memories are limited, the profilers cannot record traces of an entire system execution. SystemBuilder provides APIs to specify the timings where the profilers start and end. Designers write the API calls in any point of the process descriptions and get traces during the period they are interested in.

4.3 The Process Profiler

The process profiler records a trace of activation/wait timings of processes through the execution period specified by a designer.

Figure 3 illustrates the overall structure of our profilers. The process profiler consists of processes which are instrumented for profiling and "process profiler module" hardware (illustrated in Fig. 3). The process profiler module consists of "process trace extractor", "process trace writer", a timer module, a FIFO and a memory module. At runtime of the system, processes send signals to registers of the trace extractor. The process trace extractor collects the values of the registers, and sends them with a time-stamp obtained from the timer module

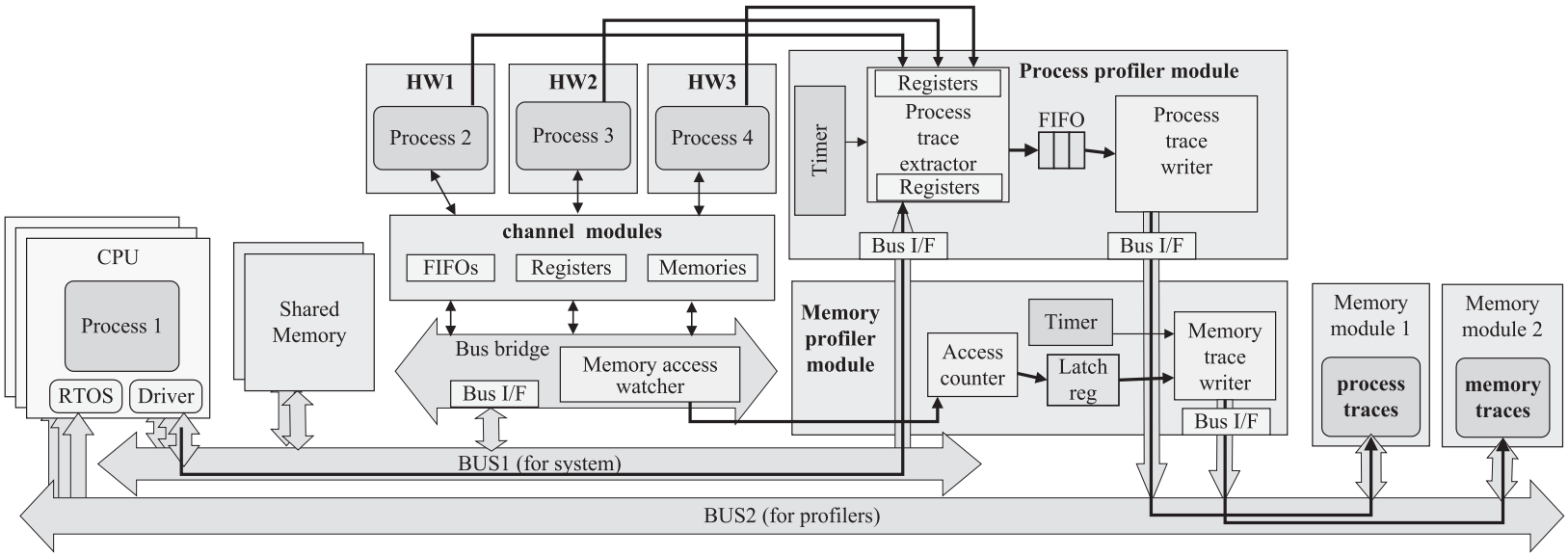


Fig. 3 Overall structure of the process profiler and the memory profiler.

to the process trace writer through the FIFO. The process trace writer writes data to the memory module whenever it receives data from the FIFO. Since we made a dedicated memory module for the process profiler and a dedicated access interface for the memory module, the memory accesses of the process trace writer do not conflict with other communications among processes, and have no effect on performance of the system.

All accesses for blocking channels, which are used to activate processes, are automatically transformed to send signals to the process trace extractor by SystemBuilder. **Figure 4** shows an example of a transformed description which accesses a blocking channel (denoted as XXX_BC_READ). The transformed description consists of an original functionality which accesses the blocking channel (denoted as function calls of `yx_meschan_read()` and `syscall()` in Fig. 4) and signaling functionalities (denoted as two function calls of `profiler_set_state()`). By the calls of `profiler_set_state()`, the process writes "0" to a register of the process trace extractor at the beginning of the access, and writes "1" at completion of the access. The description in C is converted for a target processor and an FPGA by compilers and behavioral synthesis tools, respectively.

Note that computational results of the processes do not change between before and after instrumentation of the process profiler. This is because the instrumentation of the process profiler only adds the signaling functionality to processes. However, instrumentation of the signaling functionalities can result in degradation of performance and accuracy of FPGA-prototypes. In other words, execution time (i.e., execution cycles) of processes may increase by insertion of a process profiler, and therefore, the execution cycles of the processes with the profiler are not as completely same as the ones without the profiler.

Degradation of performance is inevitable for processes implemented in software (software processes, hereafter). Software processes should notify their states to the process profiler module with additional instructions since internal states of processors are not observable from other modules. **Figure 5** shows the program code of the signaling function described in C for software processes, which appears

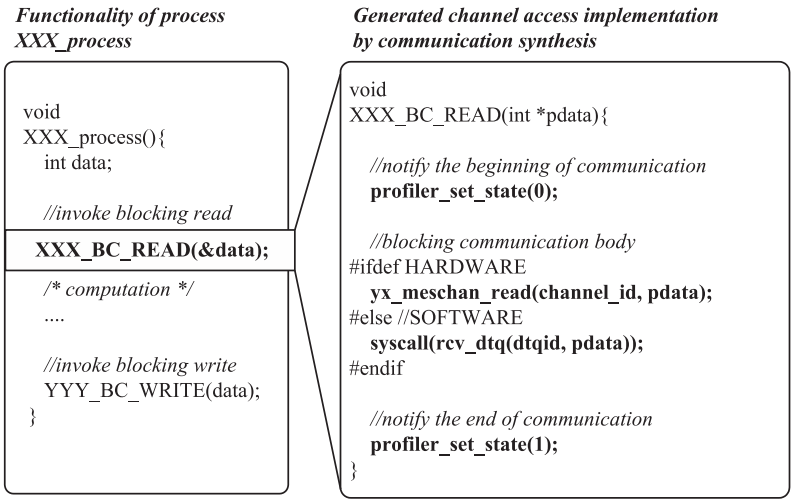


Fig. 4 Instrumentation example of a process for profiling.

```
 1: void profiler_set_state(unsigned char state){
 2:     int tskid;
 3:     unsigned char *reg4tskid;
 4:
 5:     //RTOS-API to get Task (Process) ID
 6:     get_tid(&tskid);
 7:
 8:     //get register address for this process
 9:     reg4tskid = TaskID2StateRegMap[tskid];
10:
11:     //signal state of the process
12:     //to the process trace extractor
13:     *(volatile unsigned char *)reg4tskid = state;
14: }
```

Fig. 5 An example of the signaling function for software.

as "`profiler_set_state()`" in Fig. 4. In the function, a software process, which is implemented as a task of an RTOS, first obtains its task ID using an RTOS's API. Next, the process obtains an address of a register in the process trace extractor by using the task ID. Finally, the process writes its state (executing (1) or waiting (0)) to the register. Assembly code of this function for a Nios II processor consists of 50 instructions. Since the signaling function is called twice per blocking channel access, software processes need additional 100 instructions for each blocking channel access.

As for the processes implemented in hardware (hardware processes, hereafter), "`profiler_set_state()`" in Fig. 4 is done in a single clock cycle. This is because the implementation of the `profiler_set_state()` is realized by writing a state of the process to a register in a single clock cycle. Therefore hardware processes basically need additional two clock cycles for every blocking channel access. However, the signaling functionality and the original behavior of the process may be executed in parallel (depending on behavioral synthesis results). Consequently the overhead of the signaling functionality of hardware processes will be two or less clock cycles per blocking channel access.

After the execution, the trace data are read from the memory module and are output to a debug console on a host PC. The process profiler also provides the trace analyzer & visualizer for the traces obtained from an FPGA (illustrated in Fig. 2). The trace analyzer & visualizer generates a VCD (Value Change Dump) file and a CSV (Comma Separated Value) file. The VCD file can be visualized as waveforms using tools such as *GTKWave* [18]. In the waveforms, high states mean executions of the processes and low states mean waiting times of them. Visualization of the system behavior can support designers to intuitively grasp complex parallelism of the processes. The CSV file contains formatted traces and can be fed by various tools for further analysis.

4.4 The Memory Profiler

The memory profiler records traces of shared memory accesses including access cycles and blocked cycles. Since the memory accesses are performed frequently and tend to cause exhaustion of memories for the traces, the memory profiler records the sum of the access/blocked cycles for every n cycles specified by designers in order to use limited memory capacity efficiently.

The recording part of the memory profiler is implemented in hardware (illustrated as "memory profiler module" in Fig. 3). We designed the memory profiler focusing the feature that all processes mapped on hardware are implemented to use interface for outside memories by SystemBuilder (illustrated as "bus bridge"

in Fig. 3). In order to record memory accesses, "memory access watcher" inside the bus bridge tells the occurrence of memory accesses to "access counter". The access counter records the sums of the access/blocked cycles of individual channels in a certain period, and sends the sums to "memory trace writer". The period is specified by designers using an API at the beginning of profiling. For each period, the memory trace writer sends the sums of access/blocked cycles to the dedicated memory module which stores the memory access traces. SystemBuilder automatically configures the memory access watcher depending on mapping of memory channels onto shared memory modules, and instruments the hardware module with the memory profiler module.

After the profiling, the traces are read from the memory module and are transferred to a host PC. The trace analyzer & visualizer transforms the traces and generates various intuitive graphs which show statistical values of entire execution and periodic changes.

Here, we discuss the accuracy of the system between before and after instrumentation of the memory profiler. First, it should be noted that instrumentation of the memory profiler by SystemBuilder does not change implementation of processes. Therefore computational results of processes are not changed by the instrumentation. Moreover, the memory profiler does not change cycle-level behavior of the FPGA-based prototype. The memory profiler capability consists of three modules, i.e., the memory profiler module, a memory module to store memory access traces and the bus bridge where the memory access watcher is embedded. None of the three modules changes cycle-level behavior of the prototype as follows.

The access counter in the memory profiler module does not block the behavior of the bus bridge since it only receives signals brought by the memory access watcher.

The memory trace writer in the memory profiler module and the dedicated memory module ("Memory module 2" in Fig. 3) are connected by a dedicated bus (denoted as "BUS2"), hence communications between them do not conflict with other communications among processes on "BUS1".

The memory access watcher actually only brings internal signals of the bus bridge to the access counter, so that it does not interfere with the cycle-level behavior of the bus bridge.

For these reasons, cycle-level behavior of FPGA-based prototypes is not changed by instrumentation of the memory profiler.

However, the number of clock cycles required to execute overall the system will increase because a software process needs to call APIs which start/stop the execution of the profiler. Nevertheless, cycle-level behavior of the system between start and stop of the memory profiler, in which designers are interested, are accurate between before and after the instrumentation.

Currently the memory profiler can trace the memory accesses performed by the processes implemented on hardware. Designers therefore cannot analyze impacts of conflicts caused by the processes on software directly, while the designers can see the impacts from the blocked cycles recorded by the memory profiler which include conflicts caused by software. This is because we laid emphasis on automation of the profiling flow and compromised on limitations of a logic synthesis tool.

4.5 Further Discussion on FPGA-based Profiling

As discussed in Section 4.3 and Section 4.4, profiler instrumentation may change the performance (execution cycles) of processes. In addition, a general problem exists in our approach to FPGA-based prototyping and profiling, i.e., FPGA-based prototypes can hardly behave as same as final ASIC implementations. This problem comes from various reasons. For example, since the capacity of FPGA is generally much smaller than that of ASIC, multiple FPGAs need to be used and the interconnection delay between FPGAs arises; some IP components (such as memory) for FPGA behave differently from those for ASIC. This problem is not specific to the work presented in this paper, but arises in any FPGA-based prototyping, and hence, is out of scope of this paper. In other words, this work (i.e., SystemBuilder with the profilers) does not provide any solution to this problem.

In fact, current SystemBuilder further complicates this problem because it supports only Altera's Nios II processors and Avalon buses, which are rarely used in final ASIC implementation in industry. In order for SystemBuilder to be used for prototyping of ASIC design in industry, SystemBuilder should support processors and buses which are actually used in final ASIC implementation. To

this end, we need not only FPGA-synthesizable IPs of the processors, buses and peripherals, but also synthesis tools and software development toolkit, and integrate these IPs and tools into our SystemBuilder design environment. This work is possible, but has not been realized yet.

5. Case Studies

In order to demonstrate the effectiveness of our profilers, we show two case studies of AES encryption system design and MPEG4 decoder system design. In addition, we evaluate performance/area overhead of the profilers with the results of the two case studies.

The case studies were performed on the following environment. The systems were designed on a PC whose OS is Windows XP Professional with an Intel Core 2 Quad 2.66 GHz processor and 2 GB RAM. The target board has an Altera Stratix II FPGA with Nios II soft-core processors at 50 MHz of clock frequency. eXCite 3.2a [19] was used for behavioral synthesis. Logic synthesis and place-and-route were done by Quartus 8.1.

5.1 AES Encryption System Design

First, we demonstrate an application example of the process profiler on an AES encryption system. **Figure 6** shows the functional structure of the AES encryption system. The AES encryption system consists of six processes, *aes_main*, *encrypt*, *keyschedule*, *addroundkey*, *looppart* and *endpart*. The six processes are executable in a pipelined manner. Each process is connected to its successor process through a blocking channel. Blocking channels have enough buffers not to block sender processes. In order to evaluate performance of the system, we measured the execution time of the system by executing the encryption 100 times per evaluation.

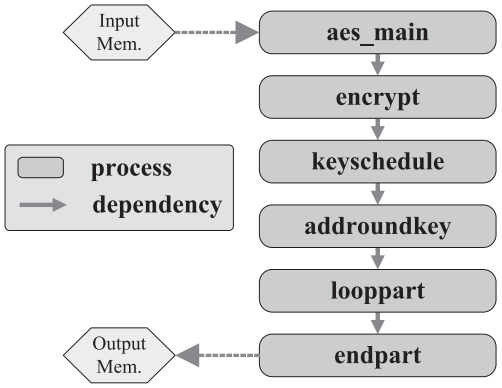


Fig. 6 Functional structure of the AES encryption system.

Figure 7 illustrates a design space exploration scenario in this case study. The purpose of this exploration is to find mapping which maximizes parallelism of the processes on a limited number of processors. We therefore explore mapping of the processes onto one or two processors, and present waveforms of processes in three different mapping specifications. The memory profiler is not needed since we do not map any process onto hardware in this case study.

As a first step, we mapped all processes on a single processor to see the execution time of individual processes (in **Fig. 8** (a)). The total execution time of the system was 2,356 milliseconds. We could see that the keyschedule and the

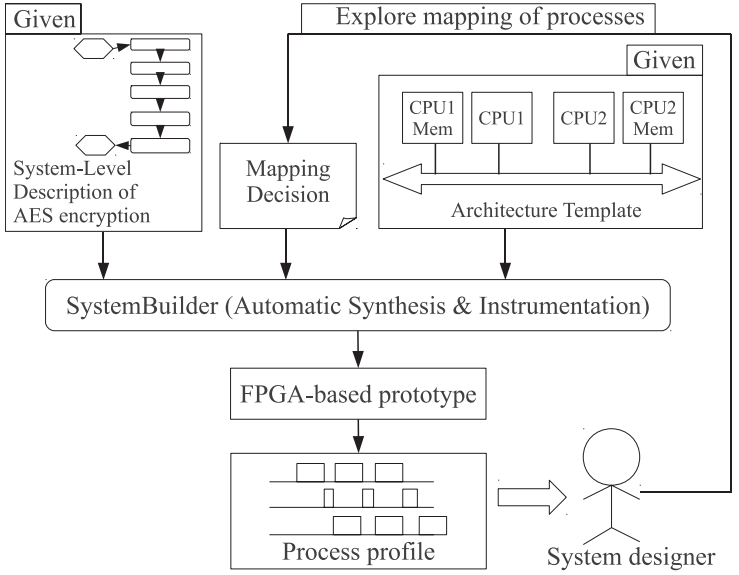
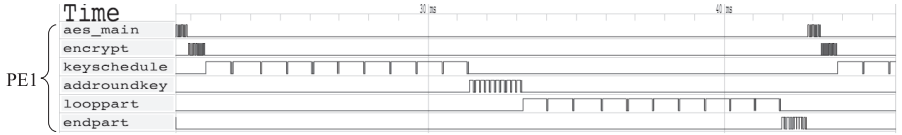
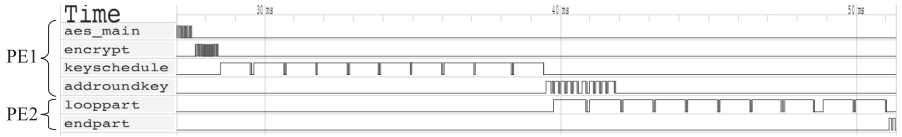


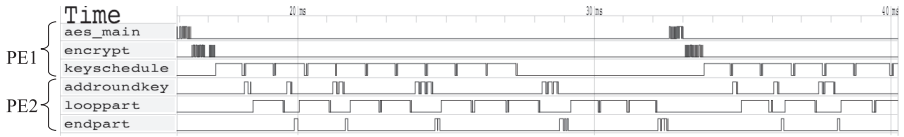
Fig. 7 Design space exploration scenario of AES design.



(a) All processes on a single processor.



(b) Last two processes on the second processor.



(c) "addroundkey" was moved to the second processor.

Fig. 8 Process profiles of AES encryption system on three mapping decisions.

looppart processes consumed significantly longer execution time than others in Fig. 8 (a). In order to improve performance, parallelizing the keyschedule process and the looppart process was expected to be obviously effective. We therefore changed mapping of the looppart and the endpart processes to the different processor.

Figure 8 (b) shows the waveforms of the processes after changing mapping of the looppart and the endpart processes. The total execution time of the system became 2,540 milliseconds. Contrary to our expectation, the total execution time was not improved and rather extended, even though the addroundkey process and the looppart processes were executed in parallel. This performance degradation might be caused by inter-processor communication. Moreover, the keyschedule process and the looppart process were not parallelized. The reason of this was obtained from Fig. 8 (b). The figure shows that the addroundkey process was not activated until the keyschedule completed its work. Since the addroundkey process was the activator of later processes, delaying activations of the addroundkey process led to the delay of the later processes, and prevented the processes from executing in parallel with the other processes. This was caused by the scheduling policy of an RTOS which the system employed. This is the good example showing that the processes whose execution time is shorter than those of others can become the bottleneck of the system and that the process profiler played an important role to find such bottleneck processes, since they cannot be found by comparing execution times of individual processes.

In order to execute the keyschedule and the looppart process in parallel, the addroundkey process must be activated immediately after an execution of the keyschedule process. It is achieved by mapping the addroundkey process onto the other processor. Figure 8 (c) is the waveforms of the processes after changing mapping of the addroundkey process. The total execution time of the system was 1,653 milliseconds. Figure 8 (c) shows clearly that the addroundkey process was activated immediately after an execution of the keyschedule process. As a result, the activation timings of the looppart process became earlier than before.

In this case study, the behavior of the processes are significantly affected by the dependencies among processes, the scheduling policy of an RTOS and the potential concurrency of processes, which cannot be considered by the analysis which relies only on the execution times of the individual processes. We could easily analyze such bottlenecks using the process traces. Thus we conclude that the effectiveness of the process profiler in exploring mapping of processes was proved.

5.2 MPEG4 Decoder System Design

We designed an MPEG4 decoder system with SystemBuilder. Based on the MPEG4 decoder developed in the past [20], we improved performance of the design using the process profiler and the memory profiler (**Fig. 9**). This case study shows the effectiveness of both the process profiler and the memory profiler on a design refinement at system level.

Figure 10 shows the structure of processes of the MPEG4 decoder system. The MPEG4 decoder consists of 12 processes, *mp4_main*, *header*, *get_mv*, *VLD*, *IQ*, *IDCT*, *catch*, *MI-1*, *MI-2*, *adder*, *yuv2rgb* and *display*. The mp4_main process handles inputs and the display process outputs decoded images to VRAM of a VGA device. The mp4_main process should be implemented in software, and the other processes can be implemented in software and hardware. So the term of "all

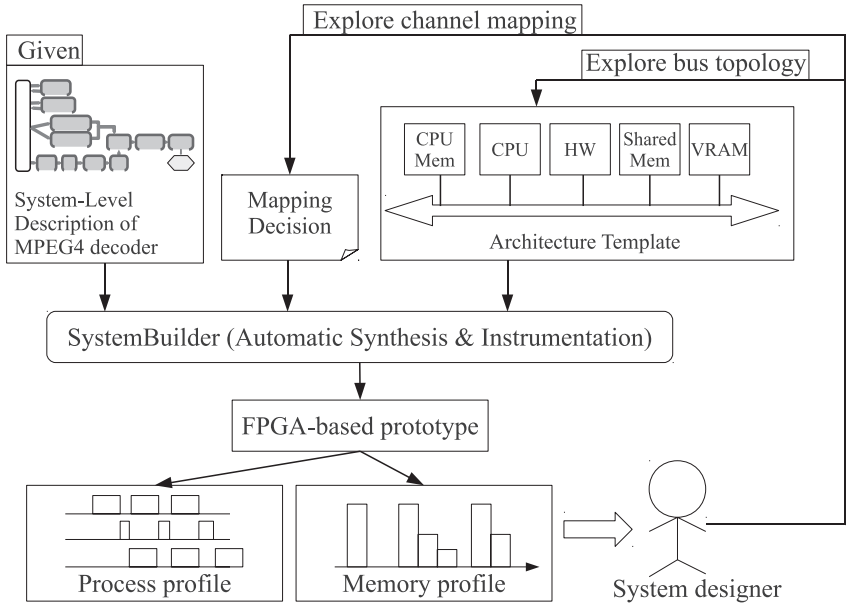


Fig. 9 Design space exploration scenario of MPEG4 decoder design.

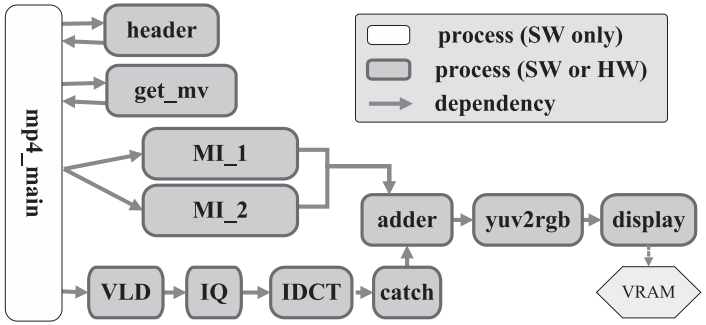
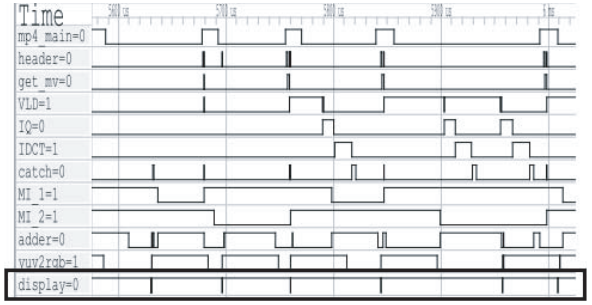
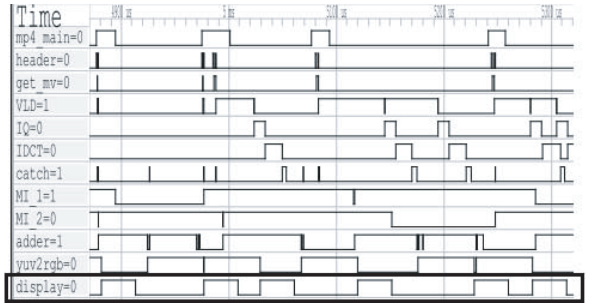


Fig. 10 Functional structure of the MPEG4 decoder.

hardware implementation" in this section denotes that all processes except for the mp4_main process are implemented in hardware. All processes can execute concurrently in a pipelined manner on all hardware implementation.



(a) MPEG4 decoder with a single bus interface.



(b) MPEG4 decoder with two bus interfaces.

Fig. 11 The process profiler results of two MPEG4 decoder implementations.

First, we explored several number of process mapping decisions and found that the all hardware implementation was the fastest implementation among them. However, its performance was yet 11.6 fps (frames per second) for 320×240 sized movies and needed further improvements. We therefore used the process profiler in order to find the bottleneck processes. **Figure 11** (a) shows the waveforms of the all hardware system. We could see that the yuv2rgb process (on the 2nd row from the bottom of the figure) could not be activated until the display process (on the bottom of the figure) finished its execution, and the display process was always active. In other words, the display process was a bottleneck. However, no solution was gained by reviewing the C program of the display process. We therefore used the memory profiler to obtain more information

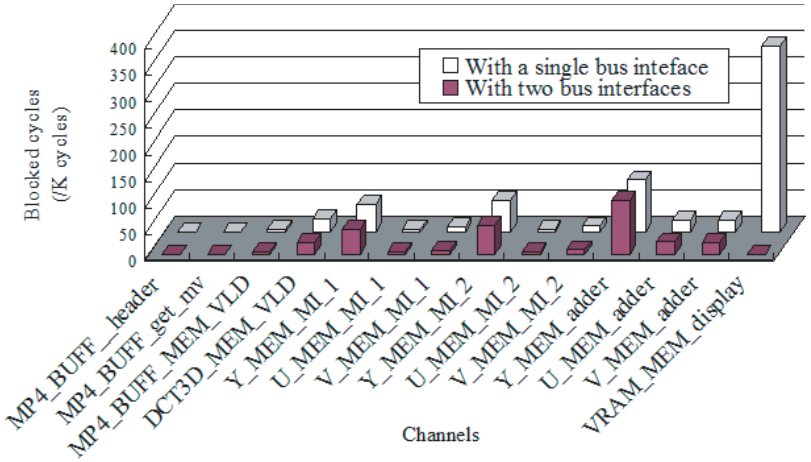


Fig. 12 Sums of blocked cycles for each channels.

about the bottleneck.

Figure 12 illustrates a graph obtained by the memory profiler. White bars on the back side of Fig. 12 illustrate the sums of blocked cycles (on y-axis) for individual channels which access off-chip memories (on x-axis) of the all hardware implementation. The rightmost bar of Fig. 12 shows that the "VRAM_MEM_display" channel, which transfers decoded images to VRAM of the VGA device, was frequently blocked by other channels. This indicated that the execution of the display process was delayed by the conflicts and that the reduction of the conflicts may make the display process faster.

There were three points at which conflicts were possible to be caused: the bus, an interface of the VRAM for the VGA device and the bus interface of the hardware module (shown as "Bus bridge" in Fig. 3) which manages bus accesses from processes mapped on hardware. Since the VRAM is accessed by the display process only, the memory accesses cannot conflict with others at the interface of the VRAM. We therefore concluded that the conflicts were caused at the bus and the bus bridge. We solved the conflicts by making a special bus bridge for the display process. Since the bus is implemented as crossbar switches in the FPGA, conflicts cannot occur at any point on accesses to the VRAM_MEM_display channel if the special bus bridge is made. As a result, the special bus bridge enabled the display process to access the VRAM exclusively. We obtained a process profile shown in Fig. 11 (b) and a memory profile shown as black bars in front side of Fig. 12. In comparison with the waveforms at the bottom of Fig. 11 (a), the waveforms of Fig. 11 (b) show that the execution time of the display process was reduced. The memory profile shown in Fig. 12 shows evidently that the blocked cycles of the VRAM_MEM_display channel were removed. In conclusion, we achieved to overcome the bottleneck on the display process with the profilers.

Table 1 Performance overheads of the profilers.

	w/o profilers	w/ process profiler	w/ memory profiler
AES (SW)	2,813 ms	3,067 ms (+9%)	-
MPEG4 (SW)	28,240 ms	30,640 ms (+9%)	-
AES (HW)	561 ms	566 ms (+1%)	560 ms (+0%)
MPEG4 (HW)	1,821 ms	1,825 ms (+1%)	1,840 ms (+1%)

Table 2 Hardware cost (ALUTs) of the profilers.

	w/o profilers	w/ process profiler	w/ memory profiler
AES (SW)	3,329	4,623	-
MPEG4 (SW)	3,329	4,623	-
AES (HW)	15,144	16,434	17,237
MPEG4 (HW)	34,680	36,002	37,932

5.3 Overhead Evaluation of The Profilers

We evaluated overheads of our profilers on performance and cost. It should be noted that the overheads of the profilers do not influence the quality of the product. The overheads, which we evaluate here, only affect FPGA-based prototypes at a design space exploration phase.

Table 1 shows the performance overheads and **Table 2** shows the area overheads on an FPGA. The first two rows of the tables show the overheads on the AES encryption system and the MPEG4 decoder system in case all processes of the systems are mapped on a single processor as software. In the case, the process profiler brought 9% increase of the execution time of the systems. The overheads of the memory profiler are not shown in the case since the memory profiler has no means if there is no process on hardware. Last two rows show overheads of the all hardware systems. Because of parallelism of hardware, the profilers had

Table 3 Required time for automatic instrumentation of the profilers.

	w/o profilers		w/ process profiler		w/ memory profiler	
	comm. syn.	HW syn.	comm. syn.	HW syn.	comm. syn.	HW syn.
AES (SW)	0.04 sec.	5 min.	0.04 sec.	6 min.	-	-
MPEG4 (SW)	0.06 sec.	5 min.	0.06 sec.	6 min.	-	-
AES (HW)	0.29 sec.	33 min.	0.18 sec.	35 min.	0.29 sec.	37 min.
MPEG4 (HW)	0.56 sec.	69 min.	0.57 sec.	74 min.	0.56 sec.	72 min.

fewer effects on performance and resulted in only 1% overhead for both systems. As for area consumption, additional ALUTs (adaptive look-up tables) are used for hardware part of the profilers. Table 2 shows that 1,200 or more ALUTs are necessary for the profilers.

We also evaluated synthesis time overhead brought by instrumentation of the process profiler and the memory profiler. The overhead on synthesis time may affect efficiency on design space exploration which is realized by iteration of changing inputs, synthesis and evaluation. **Table 3** shows synthesis time of the systems shown in Table 1 and Table 2 with and without the two profilers. Synthesis time for the systems was measured at two synthesis phases: one is communication synthesis phase performed by SystemBuilder ("comm. syn." in Table 3), and the other is behavioral and logic synthesis phase performed by YXI eXCite and Altera Quartus ("HW syn." in Table 3).

As shown in Table 3, communication synthesis by SystemBuilder completed in a second even with the profilers, while behavioral and logic synthesis time was increased by several minutes which we believe is trivial compared with the overall design time.

6. Conclusions

In system-level design, system designers describe functionalities as processes and channels, and iterates mapping of processes onto processing elements and evaluation. We proposed two profilers for FPGA-based prototypes, a process profiler and a memory profiler. The process profiler records activation/wait timings of processes. The memory profiler records access/blocked cycles of shared memory accesses.

The profilers are automatically instrumented into the system by a system-level design tool, named SystemBuilder. Automatic instrumentation of the profilers enables smooth iteration of mapping and evaluation. Since evaluation is performed by FPGA-based prototyping, designers can evaluate design candidates fast and accurately.

We demonstrated the effectiveness of the profilers through two case studies on AES encryption system design and MPEG4 decoder system design. The AES encryption system design provided an example that the process profiler played an important role to find bottlenecks caused by dependencies of the processes. The MPEG4 decoder system design presented performance refinements using the profilers considering conflicts at memory accesses.

Currently we are working for improving the memory profiler to record shared memory accesses from not only hardware but also software. Also, we want to extend the visualizing capability of the traces for more efficient analysis.

Acknowledgments This work is in part supported by STARC (Semiconductor Technology Academic Research Center).

References

1) Keutzer, K., Malik, S., Newton, A.R., Rabaey, J.M. and Sangiovanni-Vincentelli, A.: System level design: orthogonalization of concerns and platform-based design, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.19, No.12, pp.1523–1543 (2000).
2) Dömer, R., Gerstlauer, A., Peng, J., Shin, D., Cai, L., Yu, H., Abdi, S. and Gajski, D.D.: System-on-chip environment: a SpecC-based framework for heterogeneous MPSoC design, *EURASIP Journal on Embedded Systems*, Vol.2008, No.3, pp.1–13 (2008).
3) Honda, S., Tomiyama, H. and Takada H.: RTOS and codesign toolkit for multiprocessor systems-on-chip, *12th Asia and South Pacific Design Automation Conference* (*ASP-DAC*), pp.336–341 (2007).
4) Thompson, M., Nikolov, H., Stefanov, T., Pimentel, A.D., Erbas, C., Polstra, S. and Deprettere, E.F.: A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs, *International Conference on Hardware/Software Codesign and System Synthesis* (*CODES+ISSS*), pp.9–14 (2007).
5) Honda, S., Wakabayashi, T., Tomiyama, H. and Takada, H.: RTOS-centric cosimulator for embedded system design, *IEICE Trans. Fundamentals*, Vol.E87-A, No.12, pp.3030–3035 (2004).
6) Yi, Y., Kim, D. and Ha, S.: Fast and accurate cosimulation of MPSoC using trace-driven virtual synchronization, *IEEE Trans. Computer-Aided Design of Integrated*

Circuits and Systems, Vol.26, No.12, pp.2186–2200 (2007).
7) Carbon Design Systems, Inc.: http://www.carbondesignsystems.co.jp/.
8) CoWare Inc.: http://www.coware.com/.
9) Fummi, F., Loghi, M., Poncino, M. and Pravadelli, G.: A cosimulation methodology for hw/sw validation and performance estimation, *ACM Trans. Design Automation of Electronic Systems* (*TODAES*), Vol.14, No.2 (2009).
10) Mahadevan, S., Virk, K. and Madsen, J.: ARTS: A SystemC-based framework for multiprocessor systems-on-chip modelling, *Design Automation for Embedded Systems*, Vol.11, No.4, pp.285–311 (2007).
11) Wild, T., Herkersdorf, A. and Lee, G.Y.: TAPES — trace-based architecture performance evaluation with SystemC, *Design Automation for Embedded Systems*, Vol.10, No.2-3, pp.157–179 (2006).
12) Ha, S., Kim, S., Lee, C., Yi, Y., Kwon, S. and Joo, Y.P.: PeaCE: A hardware-software codesign environment for multimedia embedded systems, *ACM Trans. Design Automation of Electronic Systems* (*TODAES*), Vol.12, No.3 (2007).
13) Xilinx: http://www.xilinx.com/.
14) Altera Corporation: http://www.altera.com/.
15) Del valle, P.G., Atienza, D., Magan, I., Flores, J.G., Perez, E.A., Mendias, J.M., Benini, L. and Micheli, G.D.: A complete multi-processor system-on-chip FPGA-based emulation framework, *14th IFIP International Conference on Very Large Scale Integration* (*VLSI-SoC*), pp.140–145 (2006).
16) Nunes, D., Saldana, M. and Chow, P.: A profiler for a heterogeneous multi-core multi-FPGA system, *2008 International Conference on Field-Programmable Technology* (*FPT'08*), pp.113–120 (2008).
17) Takada, H. and Sakamura, K.: Towards a Scalable Real-Time Kernel for Function-Distributed Multiprocessors, *Proc. 20th IFAC/IFIP Workshop on Real-Time Programming* (1995).
18) GTKWave: http://intranet.cs.man.ac.uk/apt/projects/tools/gtkwave/.
19) Y Explorations, Inc.: http://www.yxi.com/index.html.
20) Shibata, S., Honda, S., Tomiyama, H. and Takada, H.: A case study of MPEG4 decoder design with SystemBuilder, *2009 International Symposium on VLSI Design, Automation & Test* (*VLSI-DAT*), pp.355–358 (2009).

(Received December 1, 2009)
(Revised February 26, 2010)
(Accepted April 14, 2010)
(Released August 16, 2010)

(Recommended by Associate Editor: *Yuichi Nakamura*)

Seiya Shibata received his B.E. degree in Information Engineering and M.S. degree in Information Science from Nagoya University in 2007 and 2009, respectively. Currently he is a Ph.D. candidate at the Graduate School of Information Science, Nagoya University. His research interests include system-level design automation and embedded systems.

Yuki Ando received his B.E. degree in Information Engineering in 2009. Currently he is a M.S. degree student at the Graduate School of Information Science, Nagoya University. His research interests include system-level design automation and embedded systems.

Shinya Honda received his Ph.D. degree in the Department of Electronic and Information Engineering, Toyohashi University of Technology in 2005. From 2004 to 2006, he was a researcher at Nagoya University Extension Course for Embedded Software Specialists. In 2006, he joined the Center for Embedded Computing Systems, Nagoya University, as an assistant professor, where he is now an associate professor. His research interests include system-level design automation and real-time operating systems. He received the best paper award from IPSJ in 2003. He is a member of IEICE and JSSST.

Hiroyuki Tomiyama received his Ph.D. degree in computer science from Kyushu University in 1999. From 1999 to 2001, he was a visiting postdoctoral researcher with the Center of Embedded Computer Systems, University of California, Irvine. From 2001 to 2003, he was a researcher at the Institute of Systems & Information Technologies/KYUSHU. In 2003, he joined the Graduate School of Information Science, Nagoya University, as an assistant professor, and became an associate professor in 2004. In 2010, he joined the College of Science and Engineering, Ritsumeikan University as a full professor. His research interests include design automation, architectures and compilers for embedded systems and systems-on-chip. He currently serves as associate editor-in-chief for IPSJ Transactions on SLDM, and associate editor for IEEE Embedded Systems Letters and International Journal on Embedded Systems. He has also served on the organizing and program committees of several premier conferences including ICCAD, ASP-DAC, DATE, CODES+ISSS, and so on. He is a member of ACM, IEEE and IEICE.

Hiroaki Takada is a professor at the Department of Information Engineering, the Graduate School of Information Science, Nagoya University. He is also the executive director of the Center for Embedded Computing Systems (NCES). He received his Ph.D. degree in Information Science from the University of Tokyo in 1996. He was a research associate at the University of Tokyo from 1989 to 1997, and was a lecturer and then an associate professor at Toyohashi University of Technology from 1997 to 2003. His research interests include real-time operating systems, real-time scheduling theory, and embedded system design. He is a member of ACM, IEEE, IEICE, and JSSST.