Automatic Communication Synthesis with Hardware Sharing for Design Space Exploration

Yuki Ando[†], Seiya Shibata[†‡], Shinya Honda[†], Hiroyuki Tomiyama[†] and Hiroaki Takada[†]
[†]Graduate School of Information Science, Nagoya University, Nagoya, Japan
[‡]Japan Society for the Promotion of Science
Email: {y_ando, shibata, honda, tomiyama, hiro}@ertl.jp

Abstract— **In this paper, we present a hardware sharing method for design space exploration of multiprocessor embedded systems. In our prior work, we had developed a system-level design tool which automatically synthesizes communications among the processes. In this work, we have extended our tool so that the tool can automatically synthesize communications which realize sharing of hardware among different processes. With the tool, designers only need to change the mapping information for hardware sharing. Designers therefore can easily explore wider design space with hardware sharing. A case study shows the effectiveness of our hardware sharing method.**

I. Introduction

System-level design has been proposed in order to design complex embedded systems. In the system-level design, designers design a system at high level of abstraction. They start from describing functionalities of the system as processes and channels, which indicate computations and communications among processes, respectively. Then they decide mapping of processes to various Processing Elements (PEs) including CPUs and dedicated hardware modules. In order to support such system-level design, a number of tools have been proposed in the past [1][2][3][4]. The tools have ability to convert processes and channels into compilable software program and synthesizable RTL circuits depending on the mapping decision.

These days, embedded systems consist of multiple applications (such as music and movie players, email and web browsing), and in many cases the applications include common functionalities (such as DCT, IDCT, encryption and decryption). In order to optimize the cost/performance efficiency, the common functionalities are often implemented in dedicated hardware modules which are shared by the applications. However, such coarse-grained hardware sharing is not supported by most of the existing system-level design tools. Many tools assume single-application systems. Some tools assume multiple applications, but they do not allow mapping processes in different applications onto a single hardware module. Even if allowed, they do not automatically synthesize interface circuitry which realizes mutually exclusive accesses to the shared hardware modules.

In this work, we have extended our system-level design tool named SystemBuilder so that it supports process-level hardware sharing. With SystemBuilder, designers can map processes in different applications onto a single hardware module. Then, SystemBuilder can automatically synthesize communications for the hardware module which are shared by the multiple applications. Since the applications may run concurrently, the interface circuit generated by SystemBuilder realizes mutually exclusive accesses to the shared hardware.

This paper is organized as follows. Section II explains a brief overview of SystemBuilder. Section III presents the detail of communication synthesis for hardware sharing. Section IV shows the effectiveness of hardware sharing through a case study on Advanced Encryption Standard (AES) system and Section V concludes this paper.

II. SystemBuilder

In this section, we show a brief overview of SystemBuilder to make this paper self contained. Please refer [5] for the detail of SystemBuilder.

Figure 1 shows the mapping and synthesis overview of SystemBuilder. SystemBuilder takes functional description, an architecture template and mapping information as input, and generates target implementations of the system. The functional description and the architecture template represent system functionalities and target platforms, respectively. The functional description consists of a set of processes running concurrently and channels representing communications among processes. Processes are written in the C language with communication APIs as interfaces to channels. A process is implemented as either a software task on a Real-time OS (RTOS) or a hardware module with a single FSM depending on mapping information on software/hardware partitioning.

SystemBuilder provides abstract communications as channels and synthesizes implementation of them. One of the features of SystemBuilder is automatic synthesis of communications among the processes. Channels are classified into two general groups, asynchronous and synchronous. Asynchronous channels are used to transfer data among the processes. Synchronous channels are mainly used between two processes to notify start/end events of execution to synchronize them. Depending on the types of channels and mapping information on software/hardware partitioning, communication APIs used in each process description are converted to interface programs and logics to communicate with each other through channels.

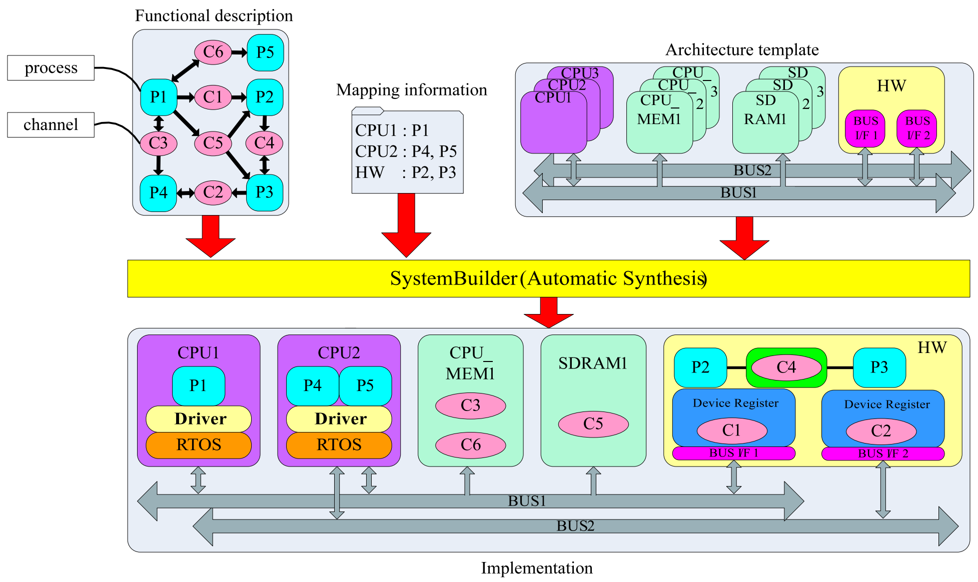

Fig. 1. Overview of SystemBuilder

III. Automatic Communication Synthesis with Hardware Sharing

A. The Design Flow for Hardware Sharing

It is assumed that a system consists of more than one application. In Fig. 2, there are two applications. Designers first design applications in the system independently as shown in (a). Without hardware sharing, SystemBuilder generates the system implementation as shown in (b).

With hardware sharing, SystemBuilder converts input description (a) to internal description (c) if both process P_B and P_Y have same functionality and they are mapped to hardware. Process P_S whose functionality is as same as both process P_B and P_Y is shared by two applications. In our hardware sharing method, channels in both application1 and application2 remain if process P_S is shared by two applications as shown in (c). Then SystemBuilder automatically generates the system as shown in (d) from internal description (c).

The hardware cost of system (d) will be less than that of system (b) since two applications share a hardware module in system (d). On the other hand, the performance of system (d) may be worse than that of system (b) since one application is blocked to use a shared hardware module while the shared hardware module is used by the other application. Therefore there is a trade-off between hardware cost and performance of the system.

SystemBuilder automatically completes the hardware sharing flow in Fig. 2 with a sharing option in the mapping information. Since designers only need to turn on the option in the mapping information to share the hardware, designers will be able to explorer wider design space in short time. Note that designers can share hardware among more than two applications although Fig. 2 only shows two applications.

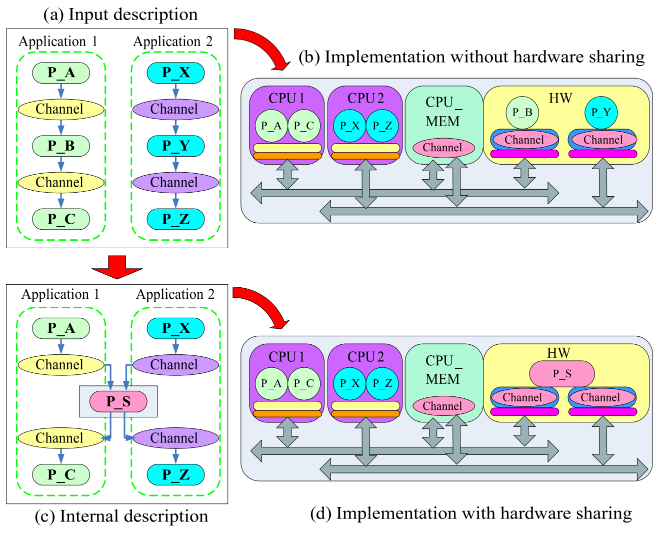

Fig. 2. Design Flow for Hardware Sharing

B. Implementation of Communication for Hardware Sharing

A shared process starts its execution by a start event of synchronous communication sent by the preceding process in the application. While the shared process is used by an application, other applications which access the shared process are forced to wait. In this way, the shared process is used by multiple applications exclusively. In our method, each application writes data to its own channels. Since applications which use the shared process have their own channels and applications use the shared process exclusively, there is no data conflict in the shared process. Instead, the shared process needs to select an application from (to) which the shared process should read (write) data.

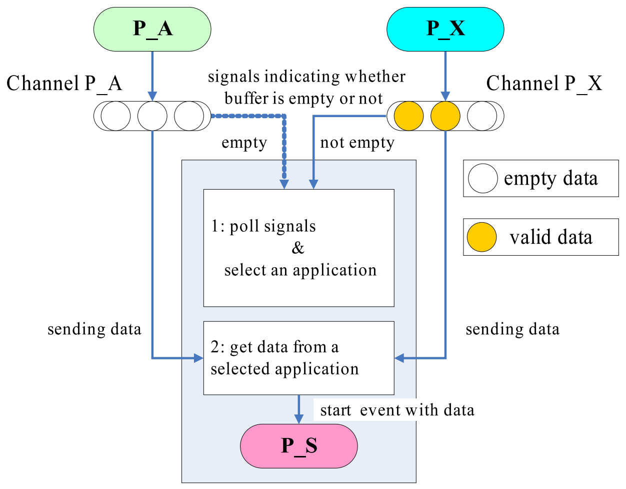
P_A
P_X
Channel P_A
signals indicating whether buffer is empty or not
Channel P_X
empty
not empty
empty data
valid data
1: poll signals & select an application
sending data
2: get data from a selected application
sending data
start event with data
P_S

Fig. 3. Detail of the Wrapper Generated by SystemBuilder

SystemBuilder automatically adds a wrapper to a shared process, which realizes mutual exclusion and identifies the channel to be accessed as shown in Fig. 3. Also, SystemBuilder adds a signal to channels connected to the shared process, which indicates if the buffer in the channel is empty or not. Then, the wrapper works as follows.

First, the wrapper polls the signals from the channels in order to select an application which can use the shared process. SystemBuilder supports two types of polling, priority-based polling and round-robin one. With priority-based polling, every time the shared process completes its execution, the channel of the highest priority application is checked at first. If the channel's signal indicates empty, the lower priority application will be checked. With round-robin polling, the channels are checked in a round-robin manner. This polling continues until non-empty signal is found. The polling type and priorities of applications are defined by designers in mapping information.

Next, data is read from the channel of the selected application, the wrapper sends a start event as well as the data to the shared process. Then, the shared process starts its execution.

The shared process may communicate with other processes not only at starting and finishing times of the process but also during its execution. Every time the shared process communicates with another process, the wrapper accesses the channel of the selected application.

It should be noted that the shared processes do not have to be modified. The wrapper generated by SystemBuilder takes care of everything needed.

C. Mapping of Processes with Hardware Sharing

In our method, there is no limitation on mapping of processes which are connected to the shared processes. Figure 4 shows four patterns on mapping of processes with hardware sharing supported by SystemBuilder. Our method can be applied to other system-level design tools if they provide a synchronous communication.

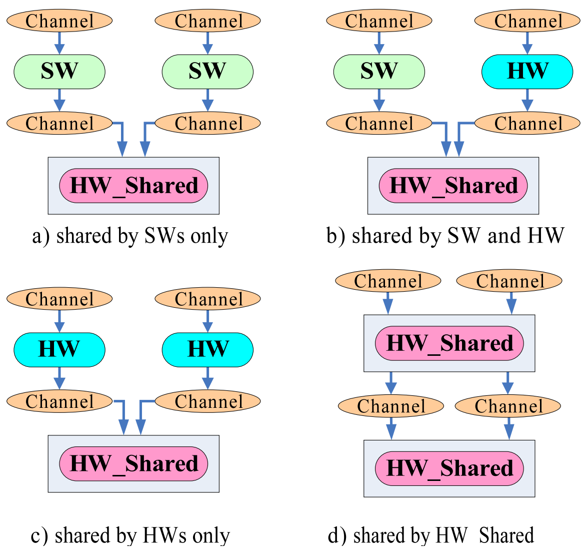

Fig. 4. Mapping of Processes with Hardware Sharing

IV. A Case Study

In this section we present a case study on an AES system to show effectiveness of hardware sharing. The AES system consists of two AES applications, AES1 and AES2. We used the AES application from CHStone Benchmark Suite [6]. Each application consists of 4 processes, aes_mainX, encryptX, decryptX and check_resultX (X is either 1 or 2).

In this case study, software processes are compiled and linked with TOPPERS/FDMP kernel [7] which is a Real-Time OS for multi-processors. Hardware processes are converted to RTL descriptions by an HLS tool, as which we used a commercial tool, YXI eXCite3.2c [8]. Hardware processes in RTL are synthesized by Quartus II 8.1 logic synthesizer and implemented on Altera Stratix II FPGA board with two Nios II soft-core processors [9]. AES1 has a higher priority than AES2 on priority-based polling. Since each application is allocated to its own processor, they can run in parallel. The performance of the AES system was measured by the time to encrypt and decrypt 16 integer data for 1000 times on two applications. Table I shows 13 designs with their process mapping, polling type, total execution time, # ALUTs, ratio of # ALUTs reduction and amount of memory usage of FPGA. # ALUTs shows only hardware area of processes mapped to hardware.

The hardware size of designs with hardware sharing was reduced by 40% compared to designs without hardware sharing, and performance of designs with hardware sharing was as same as performance of designs without hardware sharing (design 3, 4, 6 and 7). In design 10 and 11, the only difference of them is polling type. The design using round-robin polling was faster than the design using priority-based polling in total execution time. However, AES1 and AES2 took 350msec and 700msec respectively with priority-based polling while both AES1 and AES2 took 508msec with round-robin polling. With priority-based polling, the highest priority application in the system can be executed on its fastest performance while

TABLE I
EXPERIMENTAL RESULTS ON AES ENCRYPTION AND DECRYPTION

# design	mapping					performance (msec)	# ALUTs	ratio	# Memory (Mbits)
	encrypt1	encrypt2	decrypt1	decrypt2	polling				
1	SW	SW	SW	SW	—	1,941	0	—	3.83
2	HW	HW	SW	SW	—	1,236	14,180	1.00	3.93
3	HW_Share		SW	SW	priority-based	1,238	8,427	0.59	3.91
4	HW_Share		SW	SW	round-robin	1,237	8,451	0.60	3.91
5	SW	SW	HW	HW	—	725	16,417	1.00	3.93
6	SW	SW	HW_Share		priority-based	726	9,770	0.60	3.91
7	SW	SW	HW_Share		round-robin	725	9,658	0.59	3.91
8	HW	HW	HW	HW	—	350	28,985	1.00	4.01
9	HW	HW	HW_Share		priority-based	700	21,995	0.76	3.99
10	HW_Share		HW	HW	priority-based	700	23,405	0.81	3.99
11	HW_Share		HW	HW	round-robin	508	23,466	0.81	3.99
12	HW_Share		HW_Share		priority-based	700	16,432	0.57	3.97
13	HW_Share		HW_Share		round-robin	699	16,429	0.57	3.97

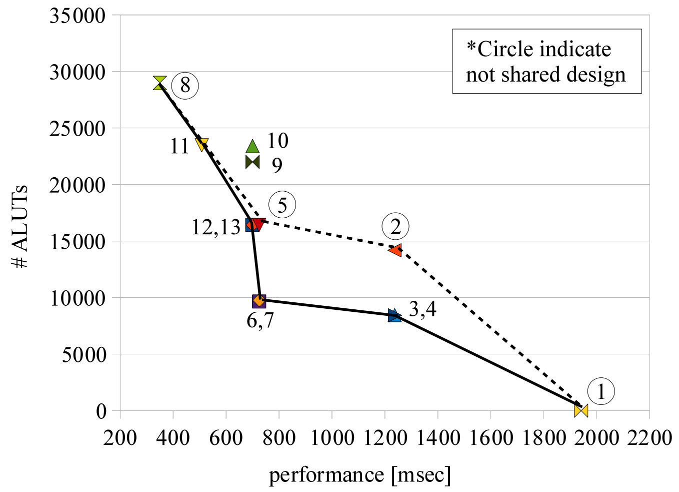

Fig. 5. Relationship Between Performance and Hardware Size

its total execution time of the system will get worse than round-robin polling one. Designers can select polling types depending on their requirements.

Figure 5 shows the plots of the relationship between performance and # ALUTs in Table I. Trade-off points of designs without hardware sharing are connected with broken lines, and trade-off points of designs with hardware sharing are connected with solid lines. The solid line indicates equal to or better performance and equal to or less hardware size than the broken line one. Hardware sharing brought better Pareto frontier. We therefore conclude hardware sharing enabled exploring better cost/performance trade-offs in this case study.

V. CONCLUSION

This paper proposed hardware sharing method with our system-level design tool named SystemBuilder. With SystemBuilder, designers only need to change mapping of processes onto either software or hardware in order to explore design space. Since SystemBuilder can automatically add the wrapper to shared process, designers only need to turn on the sharing option in mapping information in order to share the hardware. Hardware sharing will bring designers wider design spaces and chance to reduce the hardware size.

We conducted a case study of hardware sharing on an AES system which has two AES applications. In our case study, hardware sharing expanded the design space and reduced hardware size by 40% while the performance of the system did not get worse.

ACKNOWLEDGMENT

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

REFERENCES

[1] A. D. Pimentel, *The Artemis workbench for system-level performance evaluation of embedded systems*, International Journal of Embedded Systems, vol. 3, no. 3, 2008.
[2] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi and D. D. Gajski, *System-on-Chip Environment: A SpecC-Based Framework for Heterogeneous MPSoC Design*, EURASIP Journal on Embedded Systems, vol. 2008, Jan. 2008.
[3] S. Ha, S. Kim, C. Lee, Y. Yi, S. Kwon and Y. P. Joo, *PeaCE: A Hardware-Software Codesign Environment for Multimedia Embedded Systems*, ACM Trans. Design Automation of Electronic Systems, vol. 12, no. 3, Aug. 2007.
[4] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone and A. SangiovanniVincentelli, *Metoropolis: An Integrated Electronic System Design Environment*, Computer, vol. 36, no. 4, Apr. 2003.
[5] S. Honda, H. Tomiyama and H. Takada, *RTOS and Codesign Toolkit for Multiprocessor Systems-on-Chip*, ASP-DAC, Jan. 2007.
[6] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, *Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis*, Journal of Information Processing, vol.17, Oct. 2009.
[7] TOPPERS Project, http://www.toppers.jp/en/index.html.
[8] Y Explorations Inc., http://www.yxi.com/.
[9] Altera Corporation, http://www.altera.com/.