RTOS and Codesign Toolkit for Multiprocessor Systems-on-Chip

Shinya Honda Hiroyuki Tomiyama Hiroaki Takada
Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, Nagoya 464-8603, Japan
e-mail: {honda, tomiyama, hiro}@ertl.jp

Abstract— Multiprocessor designs have become popular in embedded domains for achieving the power and performance requirements. In this paper, we present principles and techniques for design and implementation of RTOS for embedded multiprocessor systems. We also present a system-level design toolkit for rapid design and evaluation of embedded multiprocessor systems.

I. Introduction

Recently, multiprocessor designs have become popular in embedded domains. This is mainly because increasing the number of processors is generally more power/performance-efficient than increasing the clock frequency. Specifically, multiprocessor systems-on-chip (MPSoCs) are considered to be a promising solution to achieve both high-performance and low-power consumption and to be used in a wide range of embedded systems in future[1, 2]. On the software side, real-time operating systems (RTOSs) have become commodity tools in order to manage the growing complexity of embedded software. In the development of embedded software running on MPSoCs, of course, RTOSs for MPSoCs are necessary.

From a viewpoint of RTOSs, there are different types of multiprocessor (MP) systems such as symmetric multiprocessor (SMP) systems, functionally distributed multiprocessor (FDMP) systems, and so on. Out of them, the FDMP architecture is an appropriate choice for embedded systems where application programs are fixed. We have developed a RTOS, named *TOPPERS/FDMP Kernel*, for FDMP-type embedded systems. In this paper, we present principles and techniques for design and implementation of TOPPERS/FDMP Kernel.

In the design of embedded multiprocessor systems, one of the most important decisions is mapping of application processes onto processors. Recent advances in EDA technologies, specifically hardware/software codesign and C-based behavioral synthesis[3, 4], further enables smooth mapping of applications not only to processors but also to hardware modules. In order to achieve the best mapping, accurate estimation of design quality such as performance and cost is necessary for each candidate mapping, but it is in general a very difficult problem. Alternatively, a designer has to evaluate the quality of mapping very quickly.

For rapid design and evaluation of embedded multiprocessor systems, we have developed a system-level design toolkit, named *SystemBuilder*. System design using SystemBuilder starts with system specification in the C language. A designer specifies the system functionalities as a set of concurrent processes communicating with each other through channels. SystemBuilder provides three primitive communication channels (communication primitives, hereafter). SystemBuilder takes the system specification and mapping directives as input, and generates RTOS-dependent software, synthesizable hardware, and hardware/software interfaces including interface circuits and device drivers. SystemBuilder also supports cosimulation at different abstraction levels and FPGA-based implementation.

This paper is organized as follows. Section II presents our RTOS for multiprocessor systems, and Section III describes our system-level design toolkit. Section IV presents a case study with a JPEG decoder application. Section V presents our on-going work, and Section VI concludes this paper.

II. Real-Time Operating Systems for Multiprocessor Systems

A. Classification of Multiprocessor Systems

There are different types of multiprocessor (MP) systems. Accordingly, RTOS supports should be different among the MP types.

MP systems are broadly classified into two types: one is tightly-coupled MP systems with shared memory, and the other is loosely-coupled MP systems (or distributed MP systems). The tightly-coupled MP systems are further classified into two types: symmetric multiprocessor (SMP) systems and functionally distributed multiprocessor (FDMP) systems (or asymmetric MP systems (AMP)). In an SMP system, every processor can access all resources (such as memory, peripherals, etc.) in the system. Therefore, an application task can be executed on any processor. In an FDMP system, on the other hand, a processor can access only a limited set of resources in the system. Therefore, an application task needs to be statically allocated to a specific processor.

In many embedded systems, only a predefined set of applications are executed. In other words, many embedded systems are dedicated to specific applications. In the design of such embedded systems, it is possible to statically map application tasks onto processors in such a way that processor loads are balanced well and/or inter-processor communication is minimized. Thus, the FDMP architecture is a natural choice for such systems in terms of cost, power consumption, and real-time performance. It should be mentioned that dynamic load balancing is still possible on an FDMP system by duplicating a task to be mapped onto more than one processor.

B. OS Supports for FDMP Systems

In many of FDMP systems so far, each processor has its own RTOS which is designed for single-processor systems, and inter-processor synchronization and communication are realized at the application level. However, this naive solution suffers from two major problems as follows:

- Programming with application-level synchronization and communication are difficult, and thus, time-consuming and error-prone.
- Exploration of task mapping alternatives[1] is not easy since application programs need to be modified every time mapping is changed.

The first problem can be solved by developing a middleware library for synchronization/communication. In order to solve the latter problem, however, OS supports are desirable such that intra-processor synchronization/communication and inter-processor synchronization/communication should be described seamlessly. In other words, OSs should provide the same APIs for both intra-processor synchronization/communication and inter-processor ones. Such OSs would not only improve the software development productivity but also facilitate rapid design space exploration to obtain the optimal solution for task mapping.

C. TOPPERS/FDMP Kernel

In the rest of this section, we describe *TOPPERS/FDMP Kernel* which we have developed for FDMP systems.

C.1. Design Principles

TOPPERS/FDMP Kernel has been developed based on *TOPPERS/JSP Kernel* which we had developed earlier for single-processor embedded systems. TOPPERS/JSP Kernel is an open-source RTOS designed as a reference implementation of *μITRON 4.0 Standard Profile*. μITRON is a standardized API for RTOSs which has been developed in Japan[5], and Standard Profile defines a fundamental subset of μITRON. Since the first release in November 2000, TOPPERS/JSP Kernel has been used in various commercial products such as printers, NC machines, and so on.

μITRON Standard Profile assumes that all application tasks are linked into a single module and all kernel objects (e.g., tasks, semaphores, etc.) are statically instantiated. RTOS service calls are implemented as normal function calls. μITRON Standard Profile does not support protection mechanism. Due to these policies, RTOSs which conform to μITRON Standard Profile can be small in size and achieve high real-time performance.

One of the most important principles in extending TOPPERS/JSP Kernel towards FDMP systems was to keep it conforming to μITRON 4.0 Standard Profile as much as possible. In other words, we avoided to change existing functionalities. Also, we tried to minimize new functionalities (service calls and APIs) to be added to μITRON Standard Profile. Thus, inter-processor synchronization/communication and intra-processor synchronization/communication can be realized with same APIs without sacrificing reusability of existing software.

[1]We consider only task mapping at the design time, and do not consider task migration at runtime.

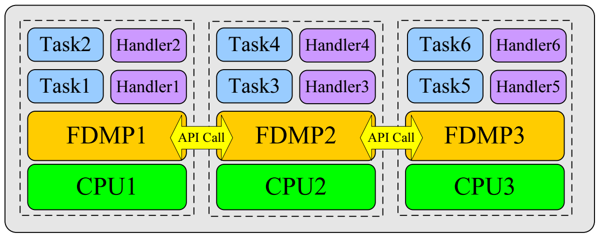

Fig. 1. An example of relationship between processors and kernel objects

C.2. Classification of Objects

In TOPPERS/FDMP Kernel, each kernel object belongs to one of the processors in the FDMP system. A set of kernel objects which belong to the same processor is called a *class*. As mentioned later, allocation of kernel objects to processors is statically determined and described in so-called a *configuration file*. In TOPPERS/FDMP Kernel, classes are, in turn, handled as kernel objects, and identification (ID) numbers are given to them. The following objects can be executed only on the processors to which the objects belong to.

- Application tasks
- Task exception handling routines
- Cyclic handlers
- Interrupt handlers
- CPU exception handlers

With TOPPERS/FDMP Kernel, a task is executable only on a specific processor, and cannot migrate between processors at runtime.

Figure 1 shows an example of relationship between processors and kernel objects.

C.3. Identification of Objects

Tasks can access all of the objects in the system using μITRON system calls, no matter which processors the objects belong to. According to the μITRON specification, unique ID numbers are given to all objects, and system calls which manipulate an object have its ID number as an argument. In TOPPERS/FDMP Kernel, an object has a unique ID number of 32 bits, and the ID number consists of two parts. The upper 16 bits are used to specify a class ID number, and the lower 16 bits are used to specify an ID number in the class. A class ID of zero means that the object belongs to the same processor on which the system call is issued. In this way, μITRON system calls can be used in TOPPERS/FDMP Kernel without changing their APIs.

C.4. System States

μITRON 4.0 defines system states for exclusively executing a specific task. Such states include *locked CPU state* where no interrupt or task switch is permitted, and *suppressed dispatch state* where no task switch is permitted. These states are often used for mutual exclusion.

In TOPPERS/FDMP Kernel, the system state is controlled processor-by-processor independently. For example, if one of the processors is in the locked CPU state, interrupts and task switches can be allowed on other processors. Therefore, mutual exclusion across processors cannot be realized by using these system states. With TOPPERS/FDMP Kernel, mutual

```
local_class CPU1{
   CRE_TSK(TASK1, {TA_HLNG, ..});
   CRE_TSK(TASK2, {TA_HLNG, ..});
   CRE_CYC(CYCHDR1, {TA_HLNG, ..});
}
local_class CPU2{
   CRE_TSK(TASK3, {TA_HLNG, ..});
   CRE_TSK(TASK4, {TA_HLNG, ..});
   CRE_CYC(CYCHDR2, {TA_HLNG, ..});
}
```

Fig. 2. A fragment of a configuration file for a dual-processor system.

exclusion should be implemented explicitly using synchronization objects (such as semaphores). If we want to reuse software with the state-based mutual exclusion mechanism, the software needs to be rewritten.

C.5. Static API

μITRON 4.0 Standard Profile defines that all kernel objects are statically instantiated. Dynamic instantiation of kernel objects at runtime is not supported. Kernel objects are defined by means of so-called static API in a configuration file. The configuration file is fed by so-called configurator to generate C files where the objects are instantiated and initialized. The C files are compiled and linked with application tasks as well as the RTOS kernel code.

In TOPPERS/FDMP Kernel, syntax of the configuration file has been extended in order to specify allocation of kernel objects to specific processors. Figure 2 shows a fragment of a configuration file for a system with two processors CPU1 and CPU2. For each processor, two tasks and one cyclic hander are instantiated. As shown in Figure 2, changing allocation of tasks and other objects is very easy. For example, if we want to reallocate TASK3 from CPU2 to CPU1, the only thing to do is to move the corresponding line from CPU2 to CPU1 in the configuration file.

The configurator creates a directory for each processor, and generates C files for instantiating the kernel objects assigned to the processor. During the process, the configurator assigns unique ID numbers for all objects. Then, for each processor, the C files are compiled and linked with applications and the RTOS kernel to generate an executable object code.

D. Evaluation

We have evaluated code size and performance of TOPPERS/FDMP Kernel. Altera NiosII/s, which is a soft-core processor for FPGA, is used as a target processor. We have developed a multiprocessor platform with four NiosII/s processors. Each processor has a local memory. The Avalon bus, which is a standard bus for NiosII systems, is based on a star-type network, so no contention happens as long as the processors access their local memories. All the components (i.e., processors, memories, bus, etc.) operate at 50MHz.

D.1. Code Size

Comparison of code size between TOPPERS/JSP Kernel and TOPPERS/FDMP Kernel is summarized in Table I. The size of the text section for the FDMP kernel is about 60% larger than that for the JSP kernel. One of the reasons for the increased code size is that, for each system call, a routine for acquiring and releasing a lock is inserted. Also, a new routine for avoiding deadlocks is added. On the other hand, an increase in the data and bss secsions is trivial. An increase in data size is also small. A new data block, named CCB (Class Control Block), of 128 bytes is added for each processor. In addition, TCB (Task Control Block) is extended by 6 bytes.

TABLE I
CODE SIZE OF JSP KERNEL AND FDMP KERNEL

Kernel	text	data	bss
JSP	26671 bytes	5 bytes	68 bytes
FDMP	42707 bytes	6 bytes	76 bytes

D.2. Performance

We have measured execution times of two system calls. One of the system calls is *wup_tak*, which wakes up a task in the wait state. We executed wup_tsk in two conditions. One condition is that the system call invokes task dispatch, and the other is that it does not. The other system call is *sig_sem*. Similar to wup_tsk, sig_sem was executed in the two conditions as described above. The key difference between wup_tsk and sig_sem is that wup_tsk aquires a task lock only, while sig_sem acquires both a task lock and an object lock.

Tables II and III show the results. The row labeled ”JSP” presents execution times of the system calls using TOPPERS/JSP Kernel. The next row ”FDMP (Intra-processor)” presents execution times in case the system calls are issued towards a task/object in the same processor using TOPPERS/FDMP Kernel. The last row ”FDMP (Inter-processor)” shows the case the system calls are issued towards a task/object in a different processor.

Compared with the JSP kernel, the execution times becomes longer even in case of inter-processor system calls. This is because of the additional routine for mutual exclusion and data structures being more complicated. In case of system calls with dispatch, the execution times of the FDMP (inter-processor) are longer than those of the FDMP (intra-processor). This is because of the increased overhead for dispatching a task on a different processor.

TABLE II
EXECUTION TIME OF SYSTEM CALL WITH TASK DISPATCH

Kernel	wup_tsk	sig_sem
JSP	5 μsec	5 μsec
FDMP ¢Intra-processor £	9 μsec	10 μsec
FDMP ¢Inter-processor £	9 μsec	10 μsec

TABLE III
EXECUTION TIME OF SYSTEM CALL WITHOUT TASK DISPATCH

Kernel	wup_tsk	sig_sem
JSP	7 μsec	6 μsec
FDMP ¢Intra-processor £	11 μsec	13 μsec
FDMP ¢Inter-processor £	17 μsec	18 μsec

III. CODESIGN TOOLKIT FOR MPSOCS

This section describes a codesign toolkit, named *SystemBuilder*, for MPSoCs. The initial version of SystemBuilder was developed for single-processor systems[7], but recently it has been significantly extended towards MPSoCs.

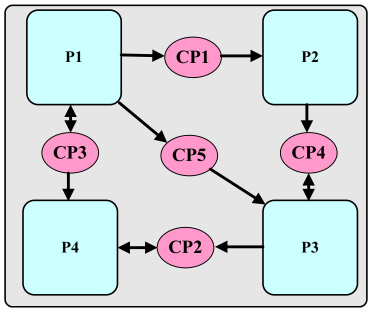

Fig. 3. An example of system description

A. Application Description

One of the inputs to SystemBuilder is an application description. In the application specification, applications are described as a set of sequential processes running concurrently. SystemBuilder provides three kinds of fundamental channels, called *communication primitives*, for communication between the processes. A process will be implemented as either a software task or a hardware module with a single FSM, depending on hardware/software partitioning.

Figure 3 shows an example of application specification where there exist four processes (denoted as P*n*) and five communication primitives (denoted as CP*n*).

Processes are written in the C language, and the overall structure is written in a specific file, named the *System DeFinition (SDF)* file.

A.1. Processes

Processes have unique names. A process may consist of multiple functions. The name of the main function of the process must be the same as the name of the process.

Processes are written in the C language. If a process might be implemented in hardware, the C code of the process must be synthesizable by a behavioral synthesis tool. At present, we use a commercial behavioral synthesis tool eXCite from YXI[6], so the coding restriction of eXCite applies to the processes on which hardware/software partitioning is not decided.

A.2. Communication Primitives

SystemBuilder supports three communication primitives: non-blocking communication primitive, blocking communication primitive, and memory primitive. In the application specification, inter-process communication must be described using the communication primitives. Each communication primitive defines access functions to use the communication primitive. Processes communicate with each other through communication primitives by calling the access functions of the communication primitives.

Communication primitives will be synthesized differently depending on hardware/software partitioning decision. This synthesis step is automated by SystemBuilder.

In order to use communication primitives in processes, the communication primitives must be declared in the SDF file. The syntax of the communication primitives is as follows, where XXX denotes a unique name given to the communication primitive.

Non-Blocking Communication Primitives

```
API
   XXX_READ(int* pdata)
   XXX_WRITE(int data)
SDF Syntax
   NBCPRIM XXX, SIZE = 8|16|32
```

Blocking Communication Primitives

```
API
   XXX_(P)READ(int* pdata)
   XXX_(P)WRITE(int data)
SDF Syntax
   BCPRIM XXX, SIZE = 8|16|32, DEPTH = xx
```

Memory Primitives

```
API
   XXX_READ(offset, int* pdata);
   XXX_WRITE(offset, data);
SDF Syntax
   MEMPRIM xxx, SIZE = 8|16|32, DEPTH = xx
```

A.3. System Definition File

The overall structure of application specification (e.g., declaration of processes and communication primitives) is described in a System DeFinition (SDF) file. In addition, mapping of processes to processing elements (such as processors and hardware modules) is specified in the SDF file.

Figure 4 shows a fragment of the SDF file for a dual-processor system with a dedicated hardware module as depicted in Figure 5. The application consists of four processes and five communication primitives. Two processes P1 and P4 are mapped to CPU1, and P2 and P3 are mapped to CPU2 and the hardware module, respectively. The SDF file also defines that CP1 is a blocking communication primitive whose bitwidth is 32 and FIFO depth is two. The five lines from BEGIN_PROCESS to END define a process named P1. P1 is written in file ”p1.c”, and three communication primitives, CP1, CP3 and CP5, are used in the process. For each communication primitive, the direction of the communication is specified. For example, CP1 and CP5 are used for write accesses, while CP3 is used for both read and write accesses.

B. Synthesis

SystemBuilder takes an SDF file and C programs as input, and automatically generates RTOS-specific software, register-transfer level (RTL) hardware, and interface between software and hardware.

B.1. Software Synthesis

Processes which are mapped to processors are translated to software tasks for TOPPERS/FDMP Kernel. Communication primitives used for intra- and inter-processor communication (such as CP1 and CP3 in Figure 5) are replaced with corresponding synchronization/communication service calls of the FDMP kernel.

B.2. Hardware Synthesis

Processes which are mapped to hardware are fed by a behavioral synthesis tool. In SystemBuilder, a commercial tool,

```
#Design Name
SYS_NAME = test
#Partition
SW(CPU1) = P1,P4
SW(CPU2) = P2
HW = P3

#Communication Primitives
BCPRIM      CP1, SIZE = 32, DEPTH = 0
BCPRIM      CP2, SIZE = 32, DEPTH = 1
...

#Processes
BEGIN_PROCESS
     NAME    = P1
     FILE    = "p1.c"
     USE_CP = CP1(OUT), CP3(INOUT), CP5(OUT)
END
...
```

Fig. 4. Fragment of the SDF file for a dual-processor system

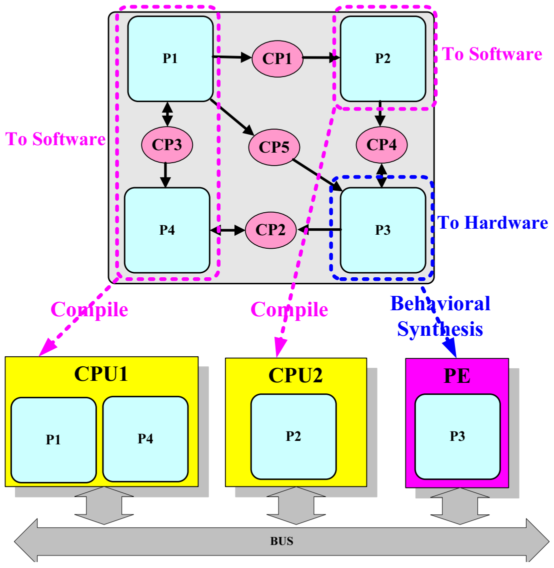

Fig. 5. A dedicated hardware module and mapping

eXCite, is used for behavioral synthesis. SystemBuilder automatically executes eXCite to generate RTL descriptions of the hardware processes. Communication primitives used for hardware/hardware communication are translated into handshaking interfaces, registers, or FIFOs, depending on the types of the communication primitives.

B.3. Interface Synthesis

Based on the SDF description, SystemBuilder automatically generate hardware/software interface. On the software side, device drivers are generated. On the hardware side, HDL descriptions of interface circuits are generated.

Hardware/software communication is based on memory-mapped I/O accesses. For each communication primitive between hardware and software, SystemBuilder instantiates a register, an FIFO or a RAM according to the communication primitive type, and assigns an address (or address space) to the storage element. Then, an address decoder circuit and an interrupt controller are synthesized. The generated circuits have an interface to a generic bus, named *VBUS*, which supports simple read/write transactions. In order to connect to an actual bus

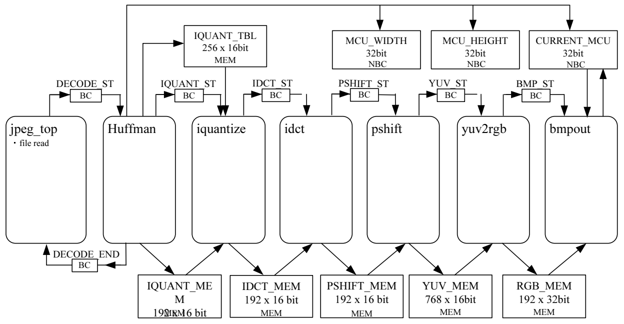

Fig. 6. JPEG decoder

such as OPB, Avalon or AMBA, a bus transducer needs to be inserted between the circuits generated by SystemBuilder and the actual bus. At present, SystemBuilder supports transducer IPs for OPB and Avalon.

On the software side, SystemBuilder generates device drivers. The device drivers include read/write access functions and interrupt handlers. Semaphores are also generated for synchronization between software and hardware.

These synthesis steps are completely automated, so a designer can explore a large number of different mappings by quick specification-synthesis-and-evaluation.

C. Cosimulation

SystemBuilder supports hardware/software cosimulation at different abstraction levels. One of the most remarkable features in our cosimulation platform is that it has a complete simulation model of μITRON-compliant RTOS so that application tasks including RTOS service calls are natively executed on a host computer. Our cosimulator also features cosimulation with functional simulation models of hardware written in C/C++ and cosimulation with HDL simulators. See [7] for more details.

IV. A Case Study

We have conducted a case study on design space exploration for a JPEG decoder application. The flow of the JPEG decoder is depicted in Figure 6. The JPEG decoder consists of seven processes, out of which four processes, i.e., *iquantize*, *idct*, *pshift*, and *yuv2rgb*, can be implemented in software or hardware. The other tasks need to be implemented in software due to the IO constraint.

In the case study, two architecture platforms were built on Xilinx Virtex-2. One is a single-processor system, and the other is a dual-processor system. The Microblaze processor is used in both platforms.

We have synthesized and evaluated 12 designs with different hardware/software partitioning on the single-processor platform. In addition, we have synthesized and evaluated 12 designs with different hardware/software partitioning and task mapping on the dual-processor platform. Figure 7 shows the cost-performance trade-offs which we have obtained through the design space exploration.

To complete the case study, it takes only a day by a single designer. It should be also noted that most time was spent for logic synthesis and place-and-route.

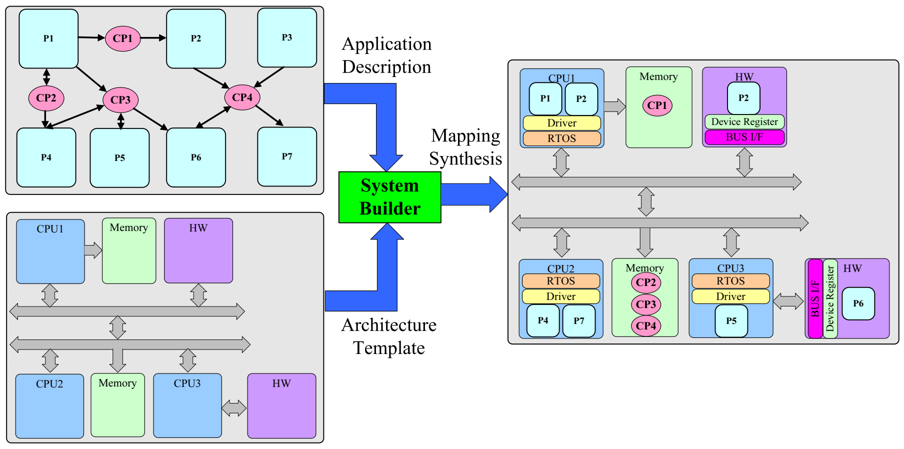

Fig. 8. Next-generation SystemBuilder design flow

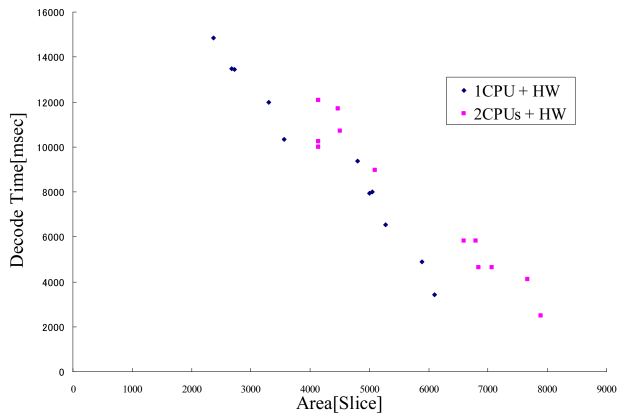

Fig. 7. Cost-performance trade-offs for JPEG decoder

V. Current Status

We have been extending SystemBuilder in two directions. One is abstract interfaces and the other is architecture exploration.

The current version of SystemBuilder supports only three communication primitives (i.e., non-blocking, blocking and memory-based channels) whose abstraction level is very low. These primitives are not sufficient at all for designers to describe their applications at high levels of abstraction. To solve this problem, we have been working on definition and synthesis of communication interfaces at higher levels of abstraction. The new interfaces include FIFOs with a sophisticated synchronization mechanism, stream data FIFOs, composite data FIFOs, shared variables with synchronization, and so on. Preliminary work on these interfaces were reported in [8].

Another ongoing work is as follows. The current version of SystemBuilder assumes that all components (such as processors, hardware modules, memories and so on) are connected to a single bus. At present, however, many multiprocessor systems have more complicated interconnections. The next-generation SystemBuilder takes as input not only an application description but also an architecture template, as shown in Figure 8. SystemBuilder will map processes and communication channels onto PEs and memories, and generate software, synthesizable hardware and interface. If the quality of the design does not meet the required level, a designer can refine his/her architecture as well as application and mapping.

VI. Conclusions

In this paper, we have described a real-time operating system, named TOPPERS/FDMP Kernel, and system-level design toolkit, named SystemBuilder, both of which we have developed for embedded multiprocessor systems-on-chip. The effectiveness of the two tools has been illustrated through a case study on design space exploration of the JPEG decoder system. We have also outlined our ongoing work on definition and synthesis of abstract interfaces as well as architectural exploration.

Acknowledgment

This work was in part supported by IPA (Information Technology Promotion Agency, Japan) and STARC (Semiconductor Technology Academic Research Center).

References

[1] S. Torii, et al, : A 600MIPS 120mW 70uA Leakage Triple-CPU Mobile Application Processor. Chip., Proc. ISSCC, 2005.

[2] T. Fujiyoshi, et.al. : An H.264/MPEG-4 Audio/Visual CODEC LSI with Module-Wise Dynamic Voltage/Frequency Scaling, Proc. ISSCC, 2005.

[3] K. Wakabayashi, "C-based Behavioral Synthesis and Verification Analysis on Industrial Design Examples," In *Proc. of Asia and South Pacific Design Automation Conference*, pp. 344–348, 2004.

[4] C. Sullivan, and A. Wilson, S. Chappell, "Using C Based Synthesis to Bridge the Productivity Gap," In *Proc. of Asia and South Pacific Design Automation Conference*, pp. 349–354, 2004.

[5] TRON Association : μITRON 4.0 Specification(Ver 4.00.00), Tokyo, JAPAN, (2002).
Avaliable at http://www.assoc.tron.org/data/spec/index-e.htm

[6] http://www.yxi.com

[7] S. Honda, H. Tomiyama, and H. Takada, "SystemBuilder: A System Level Design Environment (in Japanese)," *IEICE Trans. Information & Systems*, vol. J88-D-I, no. 2, pp. 163-174, Feb. 2005.

[8] H. Minamide, T. Yoshimoto, Y. Takagi, S. Honda, H. Tomiyama, and H. Takada, "Communication Interfaces for System Level Design," In Proc. of Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI), pp. 21-28, Nagoya, Japan, Apr. 2006.