Electronic Preprint for Journal of Information Processing Vol.28

Regular Paper

Building Fine-Grained Configurable ITRON Based RTOS

Tetsuo Miyauchi[1,a) Kiyofumi Tanaka[1,b)

Received: November 21, 2019, Accepted: May 12, 2020

Abstract: As IoT (Internet of Things) is prevailing, the number of devices which have strict resource constraints is increasing. In developing such a system, RTOS (Real Time Operating System) helps to increase productivity. However, in the view of cost reduction, it is desirable that resources for RTOS be small and the execution time be short. In this paper, we propose a method to develop an application-specific system with RTOS. Methods of removing unnecessary code for the application from RTOS kernel are explained. In addition, we implemented a reconfigurable hardware RTOS on an FPGA and applied the method for removing unnecessary code from the hardware implementation. The evaluation results show that the proposed methods reduce hardware resources, RTOS kernel execution time, and the size of the software parts in each application.

Keywords: RTOS, μITRON, configuration, system call, FPGA

1. Introduction

As IoT (Internet of Things) is prevailing, the number of devices which have strict resource constraints is increasing. Strict resource constraints are one of characteristics in developing the current embedded systems. As most of application specific systems involve development of both hardware and software, it is necessary to improve efficiency of the development to reduce the cost. RTOS (Real Time Operating System) makes it possible to easily abstract hardware, use synchronization/communication, and benefit from real-time scheduling as well as task division. Use of RTOS is effective not only in complicated systems but also in small systems [3]. However, in a cost reduction point of view, it is desirable that resources for RTOS be small and the execution time be short.

RTOS has been commonly used in various appliances for many years. RTOS system calls which are used in an application program are embedded in the executable binary code after the compilation of the application. Usually, an RTOS kernel is provided as a library format, so that only actually used system calls are linked with the application. Each RTOS system call, however, may include unused codes or unnecessary error checking codes for the application. When an application program is fixed, our objectives are: (1) removing unused codes in RTOS kernel, while leaving necessary functions such as checking possible errors, (2) implementing hardware RTOS to reduce the amount of software resources and execution time, (3) removing unused codes in hardware RTOS also, and (4) building automatic development environment with which we can perform the items above. What unnecessary codes are and how to remove them are described in Sections 3.2.1 and 3.2.2 for unnecessary codes caused by fixed attributes and the way of calling, respectively.

In order to achieve the solution, we propose the methods of automatically generating RTOS system calls which are adapted to the target application, without imposing manual insertion of directives into the application source codes on the system developers, and we implemented RTOS functions as hardware as well as software to improve the performance and reduce the footprint of software. We call hardware for executing an RTOS function "RTOS hardware", and RTOS which uses RTOS hardware "hardware RTOS". In contrast, we call RTOS of which all functions are implemented in software "software-only RTOS" in this research. This hardware RTOS works with a processor core which we developed based on MIPS instruction-set architecture. We implemented RTOS functions for error check, queue operations, getting the highest priority task and accompanying functions as RTOS hardware, which is explained in Section 3.3. We designed software-only RTOS and hardware RTOS as a subset of the standard profile in the μITRON 4.0 specification [25].

The main originality of our method is that fine-grained configuration is achieved automatically without additional directives in application source files or manual selection/deletion of code fragments. Another originality is that, along with the fine-grained adaptation, only necessary parts in system calls can be automatically built as hardware components. This paper proposes a method for configuring fine-grained RTOS based on several adaptation techniques. Similar adaptation may be done by engineers with sufficient knowledge for RTOS kernel internal structure and implementation. However, such manual adaptation would take long development time and therefore is unrealistic. Comparison between automatic adaptation, which we propose in this paper, and manual adaptation is described in Section 3.4. In addition, if some code is removed inadvertently, it may cause unexpected errors. On the other hand, our approach is available to software engineers who can implement an RTOS application system in C language with reading a software specification, which does not im-

1 Japan Advanced Institute of Science and Technology, Nomi, Ishikawa, 923–1292, Japan
a) t-miyauc@jaist.ac.jp
b) kiyofumi@jaist.ac.jp

pose knowledge about the details of RTOS or prolong the development period. With our method, we can configure fine-grained RTOS, which is implemented in hardware as well as software. Especially in hardware implementation, our hardware RTOS structure achieves one-cycle processing for queue operations.

This paper is organized as follows. Section 2 shows examples of configurable RTOS kernels and hardware implementations. In Section 3, our method for application-specific hardware and software of RTOS is presented. In addition, the structure of the RTOS hardware circuit is described. In Section 4, the evaluation environment and result are explained. Finally, in Section 5, we conclude this paper.

2. Related Work

The configuration of an RTOS kernel adapted to an application system has been studied. In the literature [9], techniques for automatically reducing the memory footprint of general-purpose operating systems on embedded platforms are described. In this literature, hand-written assembly codes in the kernel are analyzed with a decompilation technique, and unused codes and duplicated codes are eliminated. However, as the target operating system of this literature is Linux, the meaning of unused codes is different from ours. In order to build an application specific RTOS kernel, OSEK [27], which is an RTOS kernel commonly used in the automotive industry, defines OIL (OSEK Implementation Language) for description of application specific objects. This description is used as system configuration information, and a configurator generates a tailored RTOS which consists only of application specific objects and actually-used system calls. The literature [4] describes an example of OSEK-based RTOS, the main objective of which is to verify a generated RTOS, where configuration information and application codes are analyzed and an OS-application interaction graph is extracted for verification.

In order to mitigate software overhead in terms of resources and latency when RTOS is used, implementing some of the functions as hardware has been studied. SoCLC (SoC Lock Cache) hardware mechanism to improve the performance of lock latency is proposed in the literature [1]. A modular microkernel architecture in hardware is demonstrated in the literature [19]. In the literature [10], a configurable hardware scheduler architecture is presented. This scheduler provides three scheduling disciplines: priority-based, rate monotonic and earliest deadline first. This shows the advantage of modularity and the improvement of the performance. In the literature [24], three scheduler models are implemented: (i) SoRTS (Software Real-Time Scheduler), (ii) Co-SoRTS (Co-processor Software Real-Time Scheduler), and (iii) HaRTS (Hardware Real-Time Scheduler). It is concluded that Co-SoRTS and HaRTS present the best results for hard real-time applications, while SoRTS is suitable for soft real-time systems. A hardware scheduler with a new task-queue architecture to support various scheduling algorithms such as time sliced priority scheduling, Earliest Deadline First, and Least Slack Time is described in the literature [23]. RT-SHADOWS [6], [7] is a hardware scheduler and APIs to provide hardware multi-thread support, which is a subset of the task management APIs. All the hardware implementations mentioned above are only for performance improvement and do not take adaptability to a target application into account.

There are several studies for implementing a whole RTOS system call function in hardware. Silicon TRON [17] provides basic functionalities of μITRON in hardware as a peripheral chip. A real-time multithreaded operating system kernel, hthreads, is presented in the literature [2]. It has a shared memory programming model similar to POSIX Threads. In the system on CPU/FPGA chips, hardware threads and software threads can exist and they are scheduled by a hardware scheduler component, which performs first-in-first-out, round robin and priority based scheduling algorithms. In the literature [11], general purpose RTOS functions with API interfaces and a dedicated CISC processor are implemented in an FPGA. ARTESSO [12], [13] is a hardware RTOS, which provides more than 30 system calls. The specific feature of the RTOS hardware is a Virtual Queue, which is a queue structure with a tournament circuit to select an element in the queue. However, these RTOSes do not have a function to adapt to an application program automatically.

Simple and Effective hardware based Real-Time Operating System (SEOS) [20] provides adaptability for hardware RTOS. SEOS adaptation consists of hardware and software processes. However, these processes need to be applied manually. In the literature [5], OSEK-based RTOS hardware, called OSEK-V processor, is implemented with an application system after analysis of the application program, but it is not flexible to updates of the application. In the literature [21], a method of generating full hardware implementation where tasks as well as RTOS are implemented in hardware is described. To synthesize tasks for hardware, there is some restriction to tasks (e.g., no mutual exclusion). In this literature, adaptation of RTOS is not described. It is described that error checking in act_tsk takes 21 cycles while only 1 cycle is needed with our method and also unused hardware resources can be deleted in our method.

While studies for implementing RTOS functions in hardware mentioned above have been conducted for several years, we have been studying to reduce runtime and resource overhead by adapting RTOS kernel functions and processor functions to an application program. We confirmed an effect of implementing a primitive RTOS kernel operation as hardware in the literature [15]. After that, we proposed a hardware RTOS implementation in a system-call level in the literature [16]. However, automatic adaptation environment is not shown in the literature. In this paper, we explain our method for developing an application-specific system with RTOS in detail.

3. Configuration

This section describes our method to achieve the objectives listed in Section 1: (1) we show what unnecessary codes are as well as the method to generate fine-grained RTOS in Section 3.2, (2) what is implemented in hardware RTOS is described in Section 3.3, (3) our method to adapt the hardware to the application program is described in Section 3.3, and (4) we explain the detail of the automatic development environment in Section 3.4.

3.1 RTOS Structure

We have implemented a software-only RTOS kernel and one which utilizes RTOS hardware. The former does not use RTOS hardware so that all functions of RTOS work as software on a processor.

We designed software-only RTOS and hardware RTOS as a subset of the standard profile in the μITRON 4.0 specification [25]. Functions we implemented in both software-only RTOS and hardware RTOS are: Task Management Functions (act_tsk, iact_tsk, can_act, ext_tsk, ter_tsk, chg_pri), Task Dependent Synchronization Functions (slp_tsk, wup_tsk, iwup_tsk, can_wup, rel_wai, irel_wai), Semaphores (sig_sem, isig_sem, wai_sem, pol_sem), Eventflags (set_flg, iset_flg, clr_flg, wai_flg, pol_flg), and Data Queues (snd_dtq, psnd_dtq, ipsnd_dtq, fsnd_dtq, ifsnd_dtq, rcv_dtq, prcv_dtq). The other system calls in the standard profile have not been implemented yet, simply because the implementation is not in the final stage and we put priority on the system calls mentioned above. However, we can evaluate the effectiveness of our adaptation techniques with only the implemented system calls. The other ones, for example, for Fixed-Sized Memory Pool Management and Mailboxes, include similar error checking and multiple attributes, so that the same adaptation techniques can be applied.

Hardware RTOS consists of an RTOS hardware circuit and software part. RTOS functions for static error check, task status check, dynamic error check, queue operations, getting the highest priority task and changing task status are implemented in the RTOS hardware circuit and it returns the task ID of the runnable highest priority task. Since the source code of the RTOS hardware circuit is written in HDL(Verilog), the adaptation method described in the following sections can be applied to the RTOS hardware circuit as well as the software-only RTOS. Hardware RTOS has a software part as described in Section 3.3.2 and **Fig. 6**. A source file of each system call has codes for software-only RTOS and a software part of hardware RTOS, which can be selectable by a directive. Whether a software-only RTOS kernel is used or a hardware RTOS kernel is used is decided manually when the adaptation tool runs.

3.2 Adaptation Method

Since the RTOS kernel is overhead for an application program, it is desirable that resources for RTOS be small and the execution time be short. Usually, only actually used system calls are linked with an application program as an RTOS kernel is provided as a library format. Nevertheless, these system calls include unnecessary codes for the application program. We use the terms "fine-grained" and "adaptive" RTOS in this paper for an RTOS kernel in which unnecessary codes are eliminated by removing unnecessary codes caused by fixed attributes explained in Section 3.2.1 and by the way of calling explained in Section 3.2.2. Generating fine-grained RTOS is called "adaptation". In this subsection, we describe how the adaptation tool works to configure RTOS by analyzing an application and selecting functions which are actually used. To generate an application specific RTOS kernel, static APIs in the RTOS system configuration file are analyzed. In this phase, a list of IDs used in creating the specified kernel objects and the other attributes (e.g., fifo or priority order, conditions for eventflags, etc.) are extracted. This information is used both in the standard μITRON configuration process and in the proposed adaptation procedure. We explain the procedure in detail in the following sections.

3.2.1 Removing Unnecessary Codes Caused by Fixed Attributes

Some functions included in μITRON4.0 system calls are not used according to the attributes specified through parameters in a configuration file. In this case, the corresponding unnecessary code fragments can be removed from the source codes by manipulating macro descriptions.

Removing unnecessary code fragments are explained with wai_sem system call as an example below. One attribute of wai_sem system call is specified by a parameter to the static API, `CRE_SEM` described in a system configuration file. This attribute provides two options for the wait queue, fifo order and priority order which are specified by TA_TFIFO and TA_TPRI, respectively.

There are three cases: both the priority order and fifo order are used, only the priority order is used, and only the fifo order is used.

Directives for the three cases are inserted in the source codes of the system call in advance as in **Fig. 1**.

When both the priority order and the fifo order are used in the application source codes, which is used has to be determined at runtime. In this case, the code fragments for both the usages are located in the corresponding system call. When only the priority order is used in the application, the code fragment only for it is validated. Similarly, only the fragment for the fifo order is selected if it is the only usage in the application. This leads to reduction of the code size and execution time.

3.2.2 Removing Unnecessary Codes Caused by the Way of Calling

Each system call in the μITRON4.0 specification includes code fragments for checking errors. Although the μITRON4.0 specification implies that error detection can be omitted for each main error class, it may fail to notice an error which has to be detected, leading to unexpected troubles.

In the method we propose, checking codes for errors which never occur in the application are removed so that only necessary error checking exists in the object code. This is done by analyzing the application program.

Example of the Directives

```
#ifdef CHK_SEM_PRI
  if ((p_sem->sematr & TA_TPRI) != FALSE) {
    /* priority queue */
    _kernel_queue_insert_tpri( ... );
  }
#endif /* CHK_SEM_PRI */
#ifdef CHK_SEM_FIFO
  if ((p_sem->sematr & TA_TPRI) == FALSE) {
    /* FIFO queue */
    _kernel_queue_insert_prev( ... );
  }
#endif /* CHK_SEM_FIFO */
```

Fig. 1 Example of the Directives.

Table 1 System call and Error Cause (wai_sem).

System call	Error	Description	What is checked
wai_sem	E_CTX	Context error	–
	E_ID	Invalid Semaphore ID	The range of semaphore ID
	E_NOEXS	Semaphore ID Non-existent	Whether ID is created by CRE_SEM
	E_RLWAI	Forced release from waiting	rel_wai is called

The procedure of checking each system call is as follows. First, C language preprocessor is applied to an application source file to expand header files and macro definitions. Next, each call for system calls is checked and its parameters are extracted. From the parameters, possible errors at runtime are identified and the corresponding macro definitions are output. Here, since parameters originally expressed as symbolic constants defined in header files are translated into numeric values, the values are to be directly considered. On the other hand, if a parameter is given as a variable, the parameter needs to be checked at runtime, since the value of the variable is not decided statically. In this case, a macro definition that indicates the necessity of checking of the parameter at runtime is output. In addition, the information about the number of resources which is passed from the system configuration analysis is compared to the usage of the resources.

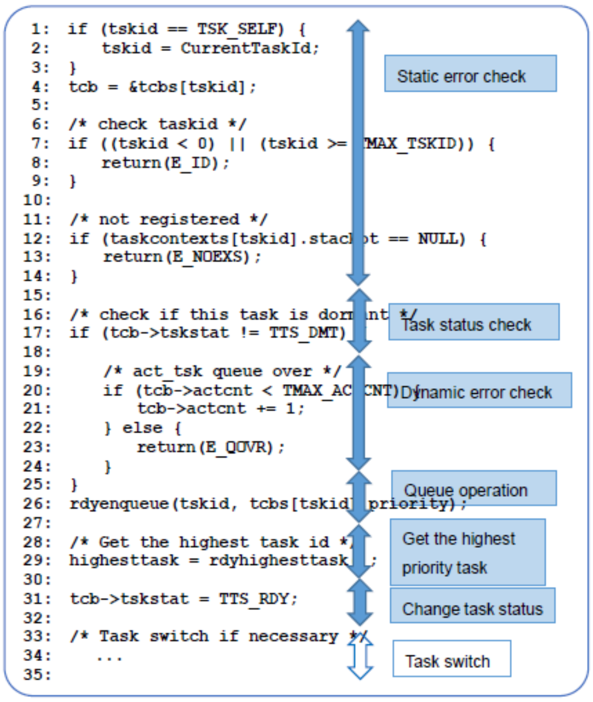
The macro definition file created by the analysis of an application, a system configuration file for μITRON4.0 convention, and RTOS kernel source codes are input to a cross compiler environment (for software-only RTOS kernel) or a hardware synthesizing tool (for hardware RTOS kernel).

Each system call in the μITRON4.0 specification has one or more possible errors and their causes. **Table 1** shows the possible errors for wai_sem system call as an example. As for the E_ID error checking, if all semaphore IDs are confirmed to be within the proper range, omitting semaphore ID checking has no effect on the behavior of the application program, so that the code of the error checking is removed from the RTOS kernel and the overhead can be reduced. For the E_NOEXS error checking, the step of the system configuration file analysis recodes IDs for created resources (semaphores) in advance of this error checking procedure. The list of these IDs is delivered to this procedure, so that all the ID values used in the application are checked and, if they are all found in the list, the code fragment for checking E_NOEXS is omitted. On the other hand, E_CTX cannot be checked statically since the error condition depends on the runtime situation of the application program, so the symbol "–" is put in the fourth column of the table. Possibility of E_RLWAI can also be statically checked.

After all errors as well as fixed attributes are checked, a hardware definition file, a file for static resource creation, and a header file are generated by the adaptation tool as shown in **Fig. 8**.

3.3 Hardware RTOS

To improve the performance and reduce the footprint of software, we implemented static error check, task status check, dynamic error check, queue operations, getting the highest priority task and changing task status as hardware while task switching is implemented as software. **Figure 2** illustrates a source code of act_tsk system call for software-only RTOS as an example. Codes indicated by blue boxes are implemented in hardware in the hardware RTOS. For the hardware RTOS, the same method as the software-only RTOS described in Sections 3.2.1 and 3.2.2 can be applied to the source code of the hardware RTOS written in HDL (Verilog), so that the hardware resources can be reduced.


```
if (tskid == TSK_SELF) {
    tskid = CurrentTaskId;
}
tcb = &tcbs[tskid];

/* check taskid */
if ((tskid < 0) || (tskid >= TMAX_TSKID)) {
    return(E_ID);
}

/* not registered */
if (taskcontexts[tskid].stackpt == NULL) {
    return(E_NOEXS);
}

/* check if this task is dormant */
if (tcb->tskstat != TTS_DMT) {

    /* act_tsk queue over */
    if (tcb->actcnt < TMAX_ACTCNT) {
        tcb->actcnt += 1;
    } else {
        return(E_QOVR);
    }
}
rdyenqueue(tskid, tcbs[tskid].priority);

/* Get the highest task id */
highesttask = rdyhighesttask();

tcb->tskstat = TTS_RDY;

/* Task switch if necessary */
   ...
```

Fig. 2 Hardware implemented part of RTOS system call (act_tsk).

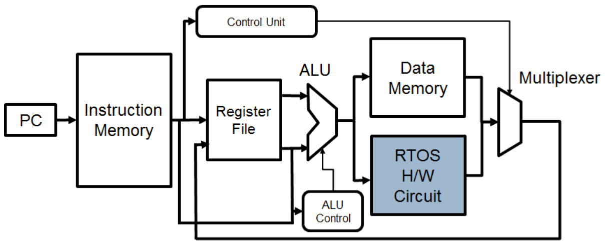


Fig. 3 Processor structure with RTOS hardware.

3.3.1 Hardware Structure

Figure 3 roughly shows a structure of the processor core and the RTOS hardware circuit we have implemented. We designed the soft processor core, of which instruction set architecture is MIPS32 [26]. RTOS hardware is accessed by memory mapped I/O.

Figure 4 depicts our RTOS hardware structure. The RTOS hardware consists of two parts, "RTOS Hardware Wrapper" and "RTOS Hardware Core". RTOS Hardware Wrapper, which is the interface between the processor core and RTOS Hardware Core, works as a state machine. When RTOS Hardware Wrapper receives an address, which indicates a command to the RTOS hardware, and data, which indicates a system call number or parameters, RTOS hardware starts to work, so that an operation such as a queue operation and input data (if any) are passed to RTOS Hardware Core.

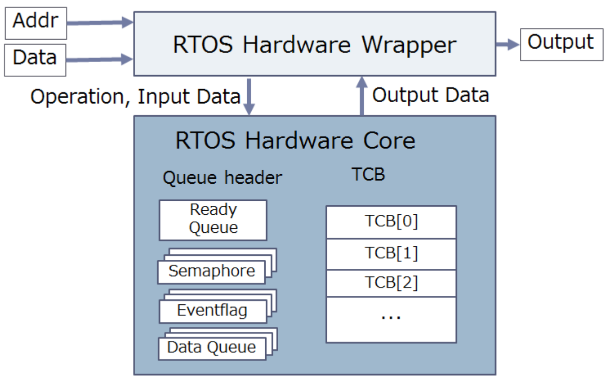


Fig. 4 Structure of RTOS hardware.

Table 2 Addresses for RTOS systemcalls.

Address	R/W	Operation
0xffff0008	R	Read RTOS return code
0xffff0100	W	Issue RTOS system call
0xffff0104	W	Set RTOS system call 1st parameter
0xffff0108	W	Set RTOS system call 2nd parameter
0xffff010c	W	Set RTOS system call 3rd parameter
0xffff0110	W	Set RTOS system call 4th parameter
0xffff0114	W	Set RTOS system call 5th parameter
0xffff0120	R	Read RTOS return parameter

3.3.2 Interface to RTOS Hardware

The structure of the system call software part is explained in this section. The software running in the processor core reads from or writes to the addresses in **Table 2**. "R" in the column "R/W" indicates that a value read from the corresponding address is a return value from the hardware. On the other hand, "W" indicates that a value is written to the address so that the value such as the system call number and other parameter values is delivered to the RTOS Hardware Wrapper.

Before the software issues a system call, it writes the parameter values to the same number of addresses (starting at 0xffff0104) as arguments defined for the system call. After all the parameters are set, the software issues the system call.

A system call is issued by writing the system call number to the corresponding address (0xffff0100). This makes the system call start by changing the state of the hardware. Then, the software reads from the address for a return code (0xffff0008) so that it checks completion of the processing and receives a task ID of the highest priority task and a return value from the system call. That is, the most significant bit of the read value indicates the completion of the RTOS hardware, and the lower bytes contain a highest-priority task ID and a return code. This is a busy-waiting procedure where, after the software writes the system call number to the address for "Issue RTOS system call" (0xffff0100), it repeatedly reads from the address for "Read RTOS return code" (0xffff0008) until it finds the most significant bit of 1. Then, it recognizes the lower bytes as a return code, and proceeds to the following processing.

Some system calls return not only a return code but the other results through call by reference. For example, wai_flg returns a flag pattern through an address which a parameter specifies. In this case, the result is obtained by reading from the address dedicated to call by reference (0xffff0120).

Figure 5 is the flow of the software part procedure for act_tsk.

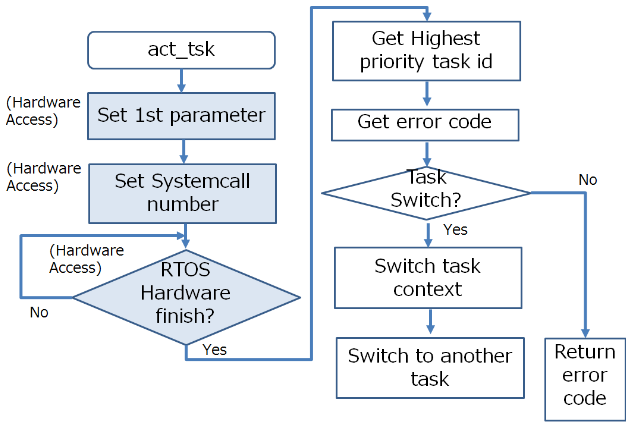


Fig. 5 System call Software flow.

act_tsk (excerpt)

```
RTOSPARAM1 = tskid;
RTOSSYSCALL = CODE_ACT_TSK;

/* When RTOS HW finish, bit:31 is on */
while (((rtosreturn = PRIHIGHEST)
          & 0x80000000) == 0);

highesttask = ((rtosreturn >> 16) & 0xff);
errorcode = (rtosreturn & 0xff);

if (errorcode == 0) {

  if (highesttask == 0) {

    /* No task switch */
    return(E_OK);
  } else {

    /* Task switch */
    RunTask(highesttask);
    return(E_OK);
  }
} else {

    /* Error case */
  return((ER)errorcode);
}
```

Fig. 6 Example of system call software part (act_tsk).

The software part code corresponding to Fig. 5 is shown in Fig. 6. Other system call functions follow a similar flow and the software part code. The software part waits for returning from hardware RTOS with polling. There is another option of using interrupt mechanisms for the completion notification. We chose polling, not interrupt, since interrupt is disabled in the system call function. In general, interrupt leads to overhead of detecting interrupt cause and context switch, while polling leads to only reading a hardware register.

3.3.3 RTOS Hardware Wrapper

RTOS Hardware Wrapper is the interface between a processor core and RTOS Hardware Core. As it works as a state machine, the next state is decided by the current state and input data. **Table 3** shows a state transition in sig_sem system call.

The state starts from INIT state after the reset. After internal registers for the hardware element are initialized in INIT state, RTOS Hardware Wrapper waits for an issue of a system call in

Table 3 State Transition (sig_sem).

State	Next State	Description
INIT	WAIT	Initialize internal registers
WAIT	CHECK	Wait for system call issued
CHECK	END	System call error occurred
	SEMHEAD	No error
SEMHEAD	SEMDEQUEUE	Check and release the first task in semaphore waiting queue
SEMDEQUEUE	END	If there is no waiting task
	RDYENQUEUE	If there is a waiting task
RDYENQUEUE	HIGHEST	Queuing a task in a ready queue
HIGHEST	END	Get the highest priority task to run
END	WAIT	Return error code

WAIT state. Software running on the processor core issues a system call by writing its parameters and then an issue signal to their corresponding addresses in Table 2.

When the RTOS Hardware Wrapper detects write access to the designated address, the state machine transits to CHECK state. In CHECK state, system call parameters are checked. When a parameter error is found, a proper error code is set in the hardware register for the return code, and the RTOS hardware transits to the END state. Software can obtain the error code by reading the register for a return parameter through the corresponding address.

When there is no error in the parameter error checking, the state transits to SEMHEAD state. In SEMHEAD state, whether there is a task queued in the semaphore waiting queue is checked and the mode transits to SEMDEQUEUE state. If there is no task queued in the semaphore waiting queue, the semaphore count is increased as long as the semaphore count does not exceed the limit, and the state transits to END state. If the semaphore count is already the maximum value of the semaphore count, E_QOVR is set to the error code and the state transits to END state. On the other hand, if there is a task queued in the semaphore waiting queue, the waiting task is released from the semaphore waiting queue and the state transits to RDYENQUEUE state.

In RDYENQUEUE state, the released task is queued to the ready queue in RTOS Hardware Core in the way described in the following subsection. As we explain later, enqueuing (i.e. registration of a TCB) is finished in one clock cycle so that it achieves large performance improvement compared to software implementation. This characteristic contributes to bounding the execution time of a system call.

After that, the state transits to HIGHEST state, in which the task ID of the highest priority in the ready queue is acquired from RTOS Hardware Core and then is written to a part of a register for return code.

Finally, the state transits to END state. When a system call returns without an error, E_OK and a task ID of the highest priority task are passed to the processor as an output from RTOS Hardware Wrapper.

3.3.4 RTOS Hardware Core

The HDL code of RTOS Hardware Core is generated by the adaptation tool according to the RTOS system configuration file. RTOS Hardware Core has TCBs and queue headers for RTOS resources such as a ready queue, semaphore waiting queues, eventflag waiting queues and data queues. RTOS Hardware Core performs operations which are requested by RTOS Hardware Wrapper. Operations are summarized by **Table 4**.

Table 4 Operations for RTOS Hardware Core.

Operation	Description
READYENQUEUE	Enqueue a TCB to a ready queue
READYDEQUEUE	Dequeue a TCB from a ready queue
PRIHIGHEST	Return a task ID of the highest priority
PRICHG	Change priority of a task
TASKSTATUS	Return a task status
SEMHEAD	Return a task ID of the top of a semaphore waiting queue.
SEMENQUEUE	Enqueue a TCB to a semaphore waiting queue
SEMDEQUEUE	Dequeue a TCB from a semaphore waiting queue
FLGHEAD	Return a task ID of the top of an eventflag waiting queue
FLGENQUEUE	Enqueue a TCB to an eventflag waiting queue
FLGDEQUEUE	Dequeue a TCB from an eventflag waiting queue
DTQHEAD	Return a task ID of the top of a data queue
DTQENQUEUE	Enqueue a TCB to a data queue
DTQDEQUEUE	Dequeue a TCB from a data queue

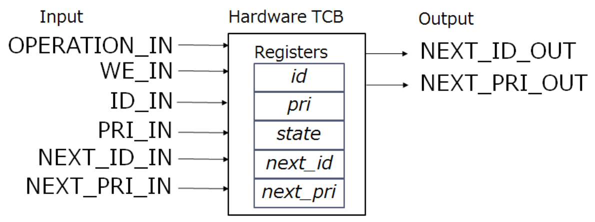


Fig. 7 TCB Structure.

Figure 7 shows a TCB structure (clock signal and reset signal are omitted). Each TCB has OPERATION_IN, WE_IN, ID_IN, PRI_IN, NEXT_ID_IN, and NEXT_PRI_IN as input data, and NEXT_ID_OUT and NEXT_PRI_OUT as output data. OPERATION_IN indicates an operation such as queuing to a priority/fifo queue or dequeuing from a queue described in Table 4. WE_IN is a write enable signal for TCB registers. ID_IN indicates a task identifier for an operation. PRI_IN shows a task priority for an operation. NEXT_ID_IN is input data for the *next_id* register. NEXT_PRI_IN is input data for the *pri* register.

TCB consists of *id*, *pri*, *state*, *next_id*, and *next_pri* registers. *id* stores task ID of this TCB. *pri* stores a priority of this TCB. The lower value is a higher priority in the μITRON specification. *state* stores a state of this TCB. *next_id* has a task ID of the next task in a queue. *next_pri* indicates a task priority of the next task in a queue.

NEXT_ID_OUT and NEXT_PRI_OUT are output data indicating the task ID and the task priority, respectively, which are used to update *next_id* and *next_pri* in a relevant TCB when a queue operation is performed. Only a TCB which becomes prior to the inserted TCB or a TCB which is being deleted generates valid values for these outputs, while the other TCBs output zeros. OR gates select the valid values and transmit them to all the TCBs through NEXT_ID_IN and NEXT_PRI_IN inputs. How a queue operation is performed with using these signals is described below.

We will show the case when tasks of *id* = 1, 2 and 4 are queued in a priority queue, and a task of *id* = 3 is designated to be enqueued in the queue. Before queuing operation, each register and signal in the TCBs are as **Table 5**. The task of *id* = 1 is the top of the queue, and the task of *id* = 2 is the next of the task of *id* = 1 since the task of *id* = 1 has *next_id* = 2. As the task of *id*

Table 5 TCB Registers and I/O (Before).

	TCB	1	2	3	4
Registers	*id*	1	2	3	4
	pri	1	3	2	5
	state	priQ	priQ	Not in Q	priQ
	next_id	2	4	0	-1
	next_pri	3	5	0	31
Input	ID_IN	X	X	X	X
	PRI_IN	X	X	X	X
	NEXT_ID_IN	X	X	X	X
	NEXT_PRI_IN	X	X	X	X
Output	NEXT_ID_OUT	0	0	0	0
	NEXT_PRI_OUT	0	0	0	0

Table 6 TCB Registers and I/O (On Queuing).

	TCB	1	2	3	4
Registers	*id*	1	2	3	4
	pri	1	3	2	5
	state	priQ	priQ	Not in Q $\rightarrow$ priQ	priQ
	next_id	2 $\rightarrow$ 3	4	0 $\rightarrow$ 2	-1
	next_pri	3 $\rightarrow$ 2	5	0 $\rightarrow$ 3	31
Input	ID_IN	3	3	3	3
	PRI_IN	2	2	2	2
	NEXT_ID_IN	2	2	2	2
	NEXT_PRI_IN	3	3	3	3
Output	NEXT_ID_OUT	2	0	0	0
	NEXT_PRI_OUT	3	0	0	0

= 4 is the last task in the queue, *next_id* = -1 and *next_pri* = 31, the maximum priority value in this configuration. In *state* row, priQ means this TCB is queued in a priority queue, and Not in Q means this TCB is not queued in any queue. X depicts this signal does not matter.

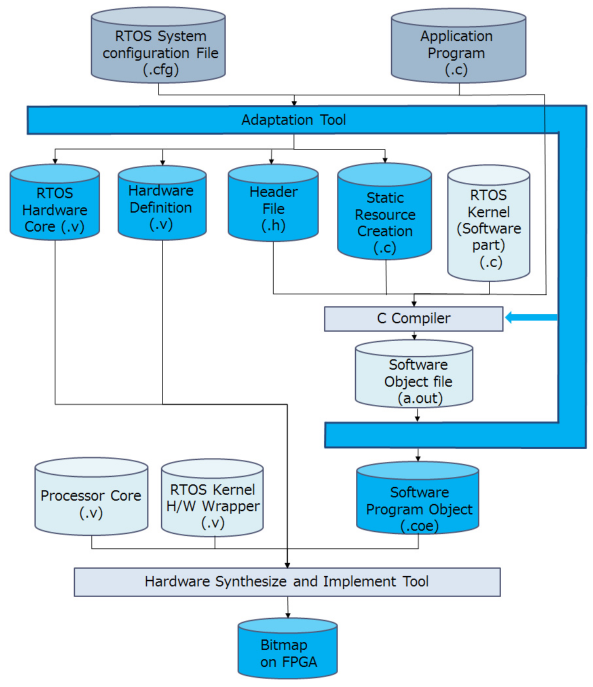
When the input signal of OPERATION_IN is READYENQUEUE, each TCB behaves as follows.

- When the input signal of PRI_IN is greater than or equal to *pri* and PRI_IN is less than *next_pri*, *next_id* and *next_pri* registers are set to the values of the input signals of ID_IN and PRI_IN, respectively. NEXT_ID_OUT and NEXT_PRI_OUT output the old values of *next_id* and *next_pri*.
- When the input signal of ID_IN is the same as *id* in the TCB, *next_id* and *next_pri* registers are set to the values of the input signals of NEXT_ID_IN and NEXT_PRI_IN, respectively, and *state* is changed to priQ.

Table 6 shows the register values, input and output signals during the enqueuing operation. *next_id* and *next_pri* in TCB of a task of *id* = 1 are set to 3 and 2, respectively, since PRI_IN = 2 is greater than *pri* in TCB of a task of *id* = 1 and lower than *next_pri*. As ID_IN = 3 and PRI_IN = 2 are input, according to the aforementioned behavior, *next_id* and *next_pri* in TCB of a task of *id* = 3 are set to 2 and 3, respectively.

3.4 Development Environment

Figure 8 shows our development environment for generating adaptive hardware and software of an RTOS. We developed an environment, of which inputs are an RTOS system configuration file and an application program. The format of the configuration file follows μITRON V4.0 specification [25], which is a collection of static system call APIs. The adaptation tool analyzes the configuration file and the application program so that an RTOS hardware core HDL (Verilog) code and hardware definitions which are used for creating adaptive RTOS hardware are



RTOS System configuration File (.cfg)
Application Program (.c)
Adaptation Tool
RTOS Hardware Core (.v)
Hardware Definition (.v)
Header File (.h)
Static Resource Creation (.c)
RTOS Kernel (Software part) (.c)
C Compiler
Software Object file (a.out)
Processor Core (.v)
RTOS Kernel H/W Wrapper (.v)
Software Program Object (.coe)
Hardware Synthesize and Implement Tool
Bitmap on FPGA

Fig. 8 Development environment.

Table 7 Comparison between automatic and manual adaptation.

Program	Automatic (sec)	Manual (sec)
semflgdtq	2.150	6,900
sem02	1.796	4,740
Cooker	1.953	4,800

generated. In addition, C language source codes for static resource creation and a header file to generate an adaptive RTOS software are created. The adaptation tool invokes a C compiler to generate a software program object from the software part of RTOS kernel source code, the header file and the static resource creation file. Finally, the hardware synthesis and implementation tool generates a bitmap for an FPGA from the software program object and the hardware HDL codes including processor core description and RTOS kernel hardware wrapper.

In order to show the performance of the automatic adaptation environment, we compared manual adaptation and automatic adaptation to three application programs, semflgdtq, sem02, and Cooker, of which the details are presented in Section 4.2. **Table 7** shows the result of our experiment. The time of Manual is the duration of manual adaptation, which was performed by a person who saw this RTOS source code at the first time and had experience of the μITRON4.0 specification so that the person can read and understand the RTOS source code. The time of Automatic is the executing time from the beginning of the adaptation process to the output of the tool, which works with Core i5-3230M 2.60 GHz CPU on Ubuntu OS. The unit of time is seconds. This result reveals that the time for adaptation takes from 4,740 seconds to 6,900 seconds with manual adaptation. In contrast, it takes from 1.796 seconds to 2.150 seconds with automatic adaptation.

4. Evaluation

This section presents the effect of our method that execution

Table 8 FPGA Resources (1).

Name	semflgdtq		semflg	sem02 [28] w/o error	flg02 [28] w/o error	dtq	Cooker [22]	Pot [22]
Adaptive	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
# of Task	5	5	5	5	5	5	4	3
# of Semaphore	4	4	4	4	0	0	0	0
# of Eventflag	3	3	3	0	3	0	1	2
# of Data Queue	3	3	0	0	0	3	1	1
Register	2,973 (16%)	2,468 (13%)	1,685 (9%)	1,430 (7%)	1,960 (10%)	2,097 (11%)	2,248 (12%)	2,117 (11%)
LUT	6,241 (68%)	4,894 (53%)	3,458 (37%)	3,179 (34%)	3,925 (43%)	4,200 (46%)	4,306 (47%)	4,322 (47%)
Slice	1,827 (80%)	1,473 (64%)	1,112 (48%)	1,024 (44%)	1,267 (55%)	1,345 (59%)	1,374 (60%)	1,377 (60%)
Minimum period	19.904 ns	19.965 ns	19.838 ns	18.264 ns	19.562 ns	18.389 ns	19.876 ns	19.956 ns
Maximum frequency	50.241 MHz	50.088 MHz	50.408 MHz	54.753 MHz	51.12 MHz	54.38 MHz	50.312 MHz	50.11 MHz

time is improved and software resources and hardware resources are reduced against software-only RTOS without adaptation.

4.1 Environment

The processor core and RTOS hardware described in the previous sections are implemented in an FPGA, Xilinx Spartan-6 (XC6SLX16CSG324C) on the evaluation board of Digilent NEXYS3, with the Xilinx development tool, PlanAhead 14.7. The processor core runs at 50 MHz and executes MIPS32 instruction set [26]. GCC 4.3.3 is used for the compiler.

We implemented an RTOS kernel, which is configured to be either a software-only kernel described in Section 3.1 or a kernel with RTOS hardware in Section 3.3. Which RTOS kernel is used is selectable when the system is configured. We evaluated the effects of application adaptation described in Section 3.2 and RTOS hardware implementation comparing with the software-only RTOS kernel without application adaptation.

4.2 FPGA Resources

Table 8 and **Table 9** illustrate the number of FPGA resources occupied by the processor core and RTOS hardware, and minimum period/maximum clock frequency for each implementation. Percentages in the parentheses indicate the rates to the whole resources. In the row of Adaptive, "Yes" shows this implementation is adapted to the application while "No" shows it is not adapted to the application.

We evaluated nine programs: "sem02" (for semaphore test) and "flg02" (for eventflag test) are from the μITRON4.0 TOPPERS kernel test suites [28], "dtq" (for dataqueue test) is our original program, "semflgdtq" and "semflg" are a combination of sem02, flg02, and dtq, and two programs, "Cooker" and "Pot", are from the literature [22] which are RTOS application programs for a rice cooker and an electric pot, respectively. In addition, "String search" and "Bit count" are benchmark programs from the literature [18], which are originally from MiBench suite [8]. (Since the two test suites programs include all error cases, intentional error checking codes are removed from the programs to evaluate the effect of adaptation). The column of w/o RTOS Hardware in Table 9 is the resource usage of only a processor. Effects of fine-grained configuration for RTOS kernel can be confirmed even with simple programs such as the ones mentioned above.

Comparing adaptive and no adaptive configurations in semflgdtq, the adaptation achieves a reduction in Register, LUT and Slice by 17%, 22% and 19%, respectively. Since semflg does

Table 9 FPGA Resources (2).

Name	String search [18]	Bit count [18]	w/o RTOS Hardware
Adaptive	Yes	Yes	–
# of Task	4	4	–
# of Semaphore	0	0	–
# of Eventflag	1	0	–
# of Data Queue	1	2	–
Register	1,697 (9%)	1,834 (10%)	843 (4%)
LUT	3,787 (42%)	4,454 (48%)	1,619 (17%)
Slice	1,288 (57%)	1,443 (63%)	529 (23%)
Minimum period	18.766 ns	17.878 ns	18.014 ns
Maximum frequency	53.288 MHz	55.935 MHz	55.512 MHz

not include a data queue system call, it can be seen that Register, LUT and Slice are reduced by 32%, 29% and 25%, respectively, compared to semflgdtq with adaptation, due to the elimination of data queue resources. The resources of sem02 show that we can further decrease Register, LUT and Slice by 15%, 8% and 8%, respectively, since it does not have the resources of eventflag. Similarly, we can see the results that the resources of flg02 and dtq are reduced due to the reduction of unused system call resources. Since Cooker employs only the necessary number of resources, Register, LUT and Slice can be reduced by 24%, 31% and 25%, respectively, compared to semflgdtq which uses more semaphores, eventflags and data queues. The same trend can be seen in the results for Pot. Finally, for String search and Bit count, considerable reduction in the resources can be seen against semflgdtq which requires more RTOS objects.

4.3 Execution Time and Size

Table 10 illustrates comparison of each system call execution time among the cases of hardware RTOS (w/ Hardware), adaptive software-only RTOS (Software w/ Adaptive), and software-only RTOS without adaptation (Software w/o Adaptive).

Since execution of system calls can involve task switching, the table includes execution times in both cases with task switching and without it. In the column of Task Switch, "No" indicates that the system call is executed and completed without task switching. Meanwhile, "Yes" corresponds to the situation where the system call execution includes task switching.

The number of clock cycles taken for executing a system call is counted and the execution time is obtained by converting the number of clock cycles to the duration of the system call execution (μsec), considering the processor core's running clock frequency of 50 MHz.

The column of w/ Hardware shows the system call execution time (μsec) of adaptive RTOS hardware with the ratio to Software w/o Adaptive case. The column of Software w/ Adaptive shows that the system call execution time (μsec) of adaptive software-only RTOS. In this case, each RTOS system call is executed on the processor without using RTOS hardware, and the ratio of Software w/ Adaptive to Software w/o Adaptive is shown. The column of Software w/o Adaptive illustrates the system call execution time (μsec) when RTOS kernel is not adapted to the application program.

From Table 10, it can be seen that, on average, w/ Hardware can reduce the execution time to 67.6% and 39.7% of the case of Software w/o Adaptive for w/o Task Switch and w/ Task Switch, respectively. For the case of Software w/ Adaptive, it is reduced to 83.4% and 94.3%, respectively. In the case of sem02 wai_sem without task switching, the execution time of w/ Hardware is longer than that of Software w/ Adaptive, the reason of which is that the semaphore count is simply implemented in this operation and there is no queue operation or priority search involved.

These results show that w/ Hardware makes the system call execution time faster than Software w/ Adaptive. Especially, the system call execution time with task switching is much reduced due to the reduction of queue operation time with hardware. On the other hand, for Software w/ Adaptive, while we can see the effect of removing unnecessary codes caused by fixed attributes and the way of calling, the rate of the reduction of the execution time is lower in the case of w/ Task Switch since it does not accelerate queue operations.

Table 10 RTOS Kernel Execution Time (μsec).

Program	System Call	Task Switch	w/ Hardware	Software w/ Adaptive	Software w/o Adaptive
sem02	sig_sem	No	1.96 (66.2%)	2.42 (81.8%)	2.96
	sig_sem	Yes	4.28 (46.5%)	8.54 (92.8%)	9.20
	pol_sem	No	1.22 (67.0%)	1.28 (70.3%)	1.82
	wai_sem	No	1.52 (77.6%)	1.28 (65.3%)	1.96
	wai_sem	Yes	4.14 (37.5%)	10.36 (93.8%)	11.04
flg02	set_flg	No	2.14 (66.9%)	2.68 (83.8%)	3.20
	set_flg	Yes	4.56 (37.8%)	10.88 (95.4%)	11.40
	pol_flg	No	2.10 (78.4%)	2.16 (80.6%)	2.68
	wai_flg	No	2.34 (78.0%)	2.48 (82.7%)	3.00
	wai_flg	Yes	4.72 (36.3%)	12.48 (96.0%)	13.0
dtq	fsnd_dtq	No	2.10 (56.1%)	3.24 (86.6%)	3.74
	fsnd_dtq	Yes	4.36 (43.8%)	9.46 (95.0%)	9.96
	psnd_dtq	No	2.10 (59.7%)	3.16 (89.8%)	3.52
	psnd_dtq	Yes	4.34 (43.9%)	9.52 (96.4%)	9.88
	rcv_dtq	No	2.14 (77.0%)	2.42 (87.1%)	2.78
	rcv_dtq	Yes	4.14 (40.9%)	9.58 (94.7%)	10.12
	prcv_dtq	No	2.14 (72.8%)	2.58 (87.8%)	2.94
Cooker	iset_flg	Yes	5.56 (43.8%)	12.16 (95.9%)	12.68
	wai_flg	Yes	4.74 (36.2%)	12.38 (94.6%)	13.08
	fsnd_dtq	Yes	4.20 (44.1%)	9.02 (94.7%)	9.52
	rcv_dtq	Yes	4.00 (39.6%)	9.56 (94.7%)	10.10
Pot	set_flg	Yes	4.48 (36.7%)	11.68 (95.6%)	12.22
	wai_flg	Yes	4.72 (36.3%)	12.32 (94.6%)	13.02
	fsnd_dtq	Yes	4.22 (42.0%)	9.54 (95.0%)	10.04
	rcv_dtq	Yes	4.02 (39.7%)	9.58 (94.7%)	10.12
String search	wai_flg	Yes	4.52 (36.7%)	11.60 (94.3%)	12.30
	fsnd_dtq	No	2.10 (58.7%)	3.08 (86.0%)	3.58
	set_flg	No	2.12 (59.2%)	3.06 (85.5%)	3.58
	set_flg	Yes	4.44 (39.9%)	10.44 (93.9%)	11.12
	rcv_dtq	No	2.12 (81.5%)	2.24 (86.2%)	2.60
Bit count	fsnd_dtq	No	2.10 (56.1%)	3.08 (82.35%)	3.74
	rcv_dtq	No	2.12 (76.3%)	2.24 (80.58%)	2.78
Average	w/o Task Switch		2.02 (67.6%)	2.49 (83.4%)	2.99
	w/ Task Switch		4.44 (39.7%)	10.54 (94.3%)	11.17

In the literature [13], it is described that RTOS execution time becomes 1.7 to 2.9 times faster in the case of w/o task switch and 1.4 to 1.5 times faster in the case of w/ task switch than software-only RTOS. The literature [20] shows that computing time is improved by 31.637% (1.4627 times faster). While the system call set in our evaluation is different from the existing studies, the system call execution time becomes 2.5–1.5 times faster on average as described below, of which results are comparable to those of other studies. In addition, there is no adaptation method to an application program in these studies. As shown in Table 10, in the case of w/o task switch, the execution time becomes 1.5 times (67.6%) faster (hardware RTOS) on average, and 1.2 times (83.4%) faster (software-only RTOS w/ adaptive) on average, and, in the case of w/ task switch, the execution time becomes 2.5 times (39.7%) faster (hardware RTOS) on average, and 1.1 times (94.3%) faster (software-only RTOS w/ adaptive) on average. Since the adaptation method explained in this paper never increases the execution time of system calls in any case, it does not have a negative effect for real-time performance com-

Table 11 RTOS Kernel Size (bytes).

Program	System Call	w/ Hardware	Software w/ Adaptive	Software w/o Adaptive
sem02	sig_sem	304 (59.4%)	400 (78.1%)	512
	pol_sem	176 (57.9%)	192 (63.2%)	304
	wai_sem	336 (48.8%)	512 (74.4%)	688
	Kernel Soft	0 (0%)	1,680 (100%)	1,680
	Others	9,104 (69.4%)	9,872 (88.9%)	11,104
	Total	9,920 (69.4%)	12,656 (88.6%)	14,288
flg02	set_flg	320 (44.4%)	608 (84.4%)	720
	pol_flg	336 (65.6%)	400 (78.1%)	512
	wai_flg	464 (44.6%)	928 (89.2%)	1,040
	Kernel Soft	0 (0%)	1,392 (100%)	1,392
	Others	10,912 (88.5%)	11,488 (93.1%)	12,336
	Total	12,032 (75.2%)	14,816 (92.6%)	16,000
dtq	fsnd_dtq	320 (41.7%)	656 (85.4%)	768
	prcv_dtq	336 (42.9%)	704 (89.8%)	848
	psnd_dtq	320 (51.3%)	544 (87.2%)	624
	rcv_dtq	368 (54.8%)	592 (88.1%)	672
	Kernel Soft	0 (0%)	1,280 (96.4%)	1,328
	Others	9,584 (85.4%)	10,512 (93.7%)	11,216
	Total	10,928 (71.0%)	14,288 (92.8%)	15,392
Cooker	iset_flg	352 (43.1%)	624 (76.5%)	816
	wai_flg	464 (44.6%)	928 (89.2%)	1,040
	fsnd_dtq	320 (41.7%)	656 (85.4%)	768
	rcv_dtq	368 (54.8%)	592 (88.1%)	672
	Kernel Soft	0 (0%)	1,824 (95.0%)	1,920
	Others	10,592 (85.5%)	11,504 (92.9%)	12,384
	Total	12,096 (68.7%)	16,128 (91.6%)	17,600
Pot	set_flg	320 (44.4%)	608 (84.4%)	720
	wai_flg	464 (44.6%)	928 (89.2%)	1,040
	fsnd_dtq	320 (41.7%)	656 (85.4%)	768
	rcv_dtq	368 (54.8%)	592 (88.1%)	672
	Kernel Soft	0 (0%)	1,824 (95.0%)	1,920
	Others	11,968 (88.1%)	12,768 (94.0%)	13,584
	Total	13,072(72.5%)	16,784(93.1%)	18,032
String search	set_flg	320 (44.4%)	576 (80.0%)	720
	wai_flg	464 (44.6%)	896 (86.2%)	1,040
	fsnd_dtq	320 (41.7%)	656 (85.4%)	768
	rcv_dtq	368 (54.8%)	592 (88.1%)	672
	Kernel Soft	0 (0%)	1,824 (95.0%)	1,920
	Others	9,888 (87.8%)	10512 (93.3%)	11,264
	Total	11,360 (69.3%)	15,056 (91.9%)	16,384
Bit count	fsnd_dtq	320 (41.7%)	656 (85.4%)	768
	rcv_dtq	368 (54.8%)	592 (88.1%)	672
	Kernel Soft	0 (0%)	1,280 (96.4%)	1,328
	Others	7,856 (87.1%)	8,464 (93.8%)	9,024
	Total	8,544 (72.5%)	10,992 (93.2%)	11,792

pared to the software-only RTOS without adaptation.

Table 11 shows the RTOS kernel sizes (bytes) of a software part of w/ Hardware, the software-only RTOS with adaptation (Software w/ Adaptive), and the software-only RTOS without adaptation (Software w/o Adaptative). For each program, the sizes of representative system calls, utility functions used in system calls (Kernel Soft), and other kernel parts (Others) are shown. Kernel Soft includes functions for queue management. Others consist of kernel initialization codes and other system calls. Percentages in parentheses indicate the rates to Software w/o Adaptive. For example, the sizes of the software part for sig_sem in sem02 is 59.4% of the size of Software w/o Adaptive.

Comparing with Software w/o Adaptive, we can see that the total sizes of the software part is reduced to 75.2% (flg02) to 68.7% (Cooker) when RTOS Hardware is used, and it is reduced to 93.2% (Bit count) to 88.6% (sem02) when software adaptation is applied.

Kernel Soft size is 0 for w/ Hardware since the corresponding operation is implemented in hardware. In the case of dtq Cooker, Pot, String search, and Bit count, Kernel Soft size is not 100% in Software w/ Adaptive, the reason of which is the test programs use only TA_TFIFO for the attribute of data queue while both of TA_TPRI and TA_TFIFO are implemented in the original (Software w/o Adaptive) Kernel Soft, so the code for the attribute of TA_TPRI is removed by adaptation. In w/ Hardware, the size of a software part of each system call is reduced to 65.6% at worst and to 41.7% at best. This large amount of reduction is achieved since most code fragments except for task switching are eliminated by adaption and Kernel Soft is fully reduced.

5. Conclusion

This paper presented a method of a development environment for application-specific systems on RTOS on an FPGA. With our environment, adaptive hardware RTOS for an application program as well as adaptive software-only RTOS can be generated. We explained our method for removing unnecessary code fragments from a hardware RTOS and a software-only RTOS. We implemented a hardware RTOS with a processor core of MIPS32 instruction-set architecture on an FPGA. The hardware RTOS circuit consists of an RTOS Hardware Wrapper and an RTOS Hardware Core. The functions of a hardware RTOS can be accessed with reading from or writing to a specific address.

We evaluated FPGA resources, RTOS kernel execution time and the size of the software parts. With proposed method, we showed that the FPGA resources and the RTOS kernel execution time were reduced in each application. Especially, for w/ Hardware, average system call execution time is reduced to 39.7% compared with Software w/o Adaptive in the case of including a task switch. From these experimental data, we conclude the proposed method is effective for developing an application-specific system with RTOS on an FPGA.

Although this paper focuses on RTOS which conforms to μITRON 4.0 specification, the proposed techniques are expected to be applied to other RTOSes. VxWorks, one of widely used RTOSes, consists of system calls which include checks of possible errors and attribute options. For example, "msgQReceive()" is a system call for message queues. This function checks several errors (not initialized, ID error, timeout, etc. [29]) at runtime and provides two options for waiting queues (FIFO and priority orders), where one of these options is selected through the function parameter [30]. It would be possible to achieve adaptation similar to those described in this paper, by elaborately analyzing application source codes and finding constant values in the parameters. For an RTOS with the μITRON4.0 specification, the method we presented in this paper can be applied to other RTOS implementations since error codes and the configuration file format are defined in the μITRON4.0 specification and if the same directives are inserted in the RTOS source code, the adaptation tool generates the adaptive RTOS. For the RTOS Hardware Wrapper, whether a similar H/W wrapper would be developed depends on a developer, but the concept we described in this paper can be applied to other implementations. We expect our approach could be a future direction for development of fine-grained configurable RTOS in commercial use.

In future work, we plan to expand this method to an application-specific multi core system with RTOS including processor core adaptation [14].

References

[1] Akgul, B.S., Lee, J. and Mooney, V.J.: A System-on-a-Chip Lock Cache with Task Preemption Support, *Proc. International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (*CASES'01*), pp.149–157 (2001).

[2] Andrews, D., Peck, W., Agron, J., Preston, K., Komp, E., Finley, M. and Sass, R.: hthreads: A hardware/software co-designed multithreaded RTOS kernel, *Proc. 2005 IEEE Conference on Emerging Technologies and Factory Automation*, pp.331–338 (2005).

[3] Anh, T.N.B. and Tan, S.-L.: Real-Time Operating Systems for Small Microcontrollers, *IEEE Micro*, Vol.29, No.5, pp.30–45 (2009).

[4] Deifel, H.P., Göttlinger, M., Milius, S., Schröder, L., Dietrich, G. and Lohmann, D.: Automatic Verification of Application-Tailored OSEK Kernels, *Proc. Formal Methods in Computer Aided Design*, pp.196–203 (2017).

[5] Dietrich, C. and Lohmann, D.: OSEK-V:Application-Specific RTOS Instantiation in Hardware, *Proc. 18th Annual ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (*LCTES*), pp.111–120 (2017).

[6] Gomes, T., Pinto, S., Garcia, P. and Tavares, A.: RT-SHADOWS: Real-Time System Hardware for Agnostic and Deterministic OSes Within Softcore, *Proc. IEEE 20th Conference on Emerging Technologies & Factory Automation* (*ETFA*), pp.1–4 (2015).

[7] Gomes, T., Garcia, P., Pinto, S., Monteiro, J. and Tavares, A.: Bringing Hardware Multithreading to the Real-Time Domain, *IEEE Embedded Systems Letters*, Vol.8, No.1, pp.2–5 (2016).

[8] Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T. and Brown, R.B.: MiBench: A free, commercially representative embedded benchmark suite, *Proc. 4th Annual IEEE International Workshop on Workload Characterization*, WWC-4, pp.3–14 (2001).

[9] He, H., Trimble, J., Perianayagam, S., Debray, S. and Andrews, G.: Code Compaction of an Operating System Kernel, *Proc. International Symposium on Code Generation and Optimization*, pp.283–298 (2007).

[10] Kuacharoen, P., Shalan, M.A. and Mooney III, V.J.: A Configurable Hardware Scheduler for Real-Time Systems, *Proc. International Conference on Engineering of Reconfigurable Systems and Algorithms*, pp.96–101 (2003).

[11] Lange, A.B., Andersen, K.H., Schultz, U.P. and Sorensen, A.S.: HartOS – a Hardware Implemented RTOS for Hard Real-Time Applications, *Proc. 11th IFAC, IEEE International Conference on Programmable Devices and Embedded Systems*, Vol.45, No.7, pp.207–213 (2012).

[12] Maruyama, N., Ishihara, T. and Yasuura, H.: An RTOS in Hardware for Energy Efficient Software-based TCP/IP Processing, *Proc. IEEE 8th Symposium on Application Specific Processors* (*SASP*), pp.58–63 (2010).

[13] Maruyama, N., Ishikawa, T., Honda, S., Takada, H. and Suzuki, K.:

ARM-based SoC with Loosely coupled type hardware RTOS for industrial network systems, *Proc. 10th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pp.9–16 (2014).

[14] Miyauchi, T. and Tanaka, K.: Configuration Technique for Adaptability of Multicore Processors on FPGA, *Proc. 27th Annual IEEE International Conference on Application-specific Systems*, Architectures and Processors, 2 pages (2016)

[15] Miyauchi, T. and Tanaka, K.: Building a Framework for an Application-Adaptive Processor System on FPGA-based SoC, *Proc. 21st Workshop on Synthesis And System Integration of Mixed Information Technologies*, pp.359–364 (2018).

[16] Miyauchi, T. and Tanaka, K.: An Adaptive Approach for Implementing RTOS in Hardware, *Proc. Embedded Systems Symposium*, pp.44–50 (2018).

[17] Nakano, T., Utama, A., Itabashi, M., Shiomi, A. and Imai, M.: Hardware Implementation of a Real-time Operating System, *Proc. 12th TRON Project International Symposium*, pp.34–42, IEEE (1995).

[18] Nikaido, Y.: Development of Benchmark Program Suite for Evaluating Embedded Operating Systems [Project Paper] Master thesis, Japan Advanced Institute of Science and Technology (2019).

[19] Nordstrom, S., Lindh, L., Johansson, L. and Skoglund, T.: Application Specific Real-Time Microkernel in Hardware, *Proc. IEEE-NPSS Real Time Conference* (*RTC*), pp.79–82 (2005).

[20] Ong, S.F., Lee, S.C., Ali, N.B.Z. and Hussin, F.A.B.: SEOS: Hardware Implementation of Real-Time Operating System for Adaptability, *Proc. 2013 1st International Symposium on Computing and Networking*, pp.612–616, IEEE (2013).

[21] Oosako, Y., Ishiura, N., Tomiyama, H. and Kanbara, H.: Synthesis of Full Hardware Implementation of RTOS-Based Systems, *Proc. 2018 International Symposium on Rapid System Prototyping*, pp.1–7, IEEE (2018).

[22] Tamura, K.: A Study on a development environment of general user interface for embedded systems, Master thesis, Japan Advanced Institute of Science and Technology (2014).

[23] Tang, Y. and Bergmann, N.W.: A Hardware Scheduler Based on Task Queues for FPGA-Based Embedded Real-Time Systems, *IEEE Trans. Computers*, Vol.64, No.5, pp.1254–1267 (2015).

[24] Vetromille, M., Ost, L., Marcon, C.A.M., Reif, C. and Hessel, F.: RTOS Scheduler Implementation in Hardware and Software for Real Time Applications, *Proc. 17th IEEE International Workshop on Rapid System Prototyping* (*RSP'06*), pp.163–168 (2006).

[25] μITRON4.0 Specification Ver.4.01.00, ITRON Committee, TRON ASSOCIATION.

[26] MIPS® Architecture For Programmers, Volume II-A: The MIPS32® Instruction Set.

[27] OSEK group, OSEK/VDX Operating System Specification 2.2.3, February 17th (2005).

[28] TOPPERS Kernel Test Suites, available from ⟨https://www.toppers.jp/testsuites.html⟩.

[29] Wind River Systems, Inc.: VxWorks Kernel API Reference, Volume 2: Routines, 6.6 (2007).

[30] Wind River Systems, Inc.: VxWorks Application Programmer's Guide, 6.2 (2005).

Tetsuo Miyauchi received his M.S. degree from Japan Advanced Institute of Science and Technology in 2015. His research interests are processor architecture, reconfigurable systems, and real-time embedded systems. He is a member of the IPSJ.

Kiyofumi Tanaka received his B.S., M.S., and Ph.D. degrees from the University of Tokyo in 1995, 1997, and 2000, respectively. His research interests are computer architecture, operating systems, and real-time embedded systems. He is a member of the IEEE, ACM, IEICE, and IPSJ.