Arrival Order Processing of Service Requests in Full Hardware Implementation of RTOS-Based Systems

Masaki Nakahara†
Graduate School of Science and Technology
Kwansei Gakuin University
Sanda, Hyogo, Japan

Nagisa Ishiura
School of Engineering
Kwansei Gakuin University
Sanda, Hyogo, Japan

***Abstract*—As an approach to drastically enhance the response performance of RTOS-based systems, Oosako proposed a full hardware scheme in which all the tasks/handlers as well as RTOS functions are implemented by hardware. In the architecture proposed so far to realize this hardware scheme, however, service requests from tasks were processed only in the order of tasks' priorities, while some RTOSs stipulate services must be processed in the order of arrival of requests. This paper proposes a hardware mechanism to enable both priority order processing and arrival order processing so that the processing order is selectively defined service by service. This is realized by newly designing an arrival order recording hardware and extending the function of the request arbitration module and flag registers to record the waiting tasks. Based on the proposed method, a management hardware that provides the functions of TOPPERS/ASP3 has been designed in Verilog HDL. The new feature has been successfully implemented, at the cost of 2.6% increase in circuit size.**

***Index Terms*—real-time systems, RTOS-based systems, full-hardware implementation, RTOS services, arrival order processing**

I. Introduction

With the recent development of information and communication technology, various services are deployed to our everyday life. Accordingly, embedded systems used in those services are required to afford higher and higher functions. In particular, control of automobile devices, unmanned aerial vehicles, and robots requires high response performance as well as rich functionality. Such systems are designed using a real-time operating system (RTOS). RTOSs provide functions to design a system to complete tasks in response to input events within predefined time periods. However, as systems become more sophisticated, it is becoming more difficult to ensure required response performance.

In order to improve the response performance of RTOS-based systems, hardware implementation of some or all of the RTOS functions has been proposed. The scheduler in RTOSs is implemented in hardware in [1] [2] [3], and most of the functions of RTOSs are implemented in hardware in [4] [5]. In these methods, however, tasks and handlers are implemented as software, which suffers from overhead due to CPU waits and context switching.

†Currently with Daifuku Co., Ltd., Japan

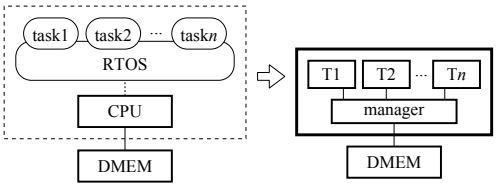


Fig. 1: Full hardware implementation [8]

To address issue, Oosako proposed a scheme to implement all tasks/handlers as well as RTOS functions as hardware [8]. The tasks are synthesized into independent hardware modules where all the tasks are executed in parallel as soon as they are ready to run. This eliminates CPU wait and task switching overheads. Ando and Muguruma proposed an improved hardware architecture for this scheme which reduces hardware size and enables the use of commercially available high-level synthesizer [12].

In this architecture, service requests from tasks were processed only in the order of tasks' priorities when multiple requests are waiting for the service. However, some RTOSs stipulates that the services must be processed in the arrival order of requests, or that users may define the processing order for each service.

This paper proposes to introduce a mechanism to enable arrival order processing into the architecture of [12]. It allow users to select between priority or arrival order processing for each service of RTOSs. The mechanism consists of decoupling of arbitration of service requests and dequeueing of waiting services, and design of a hardware module to record the arrival order of service requests.

As a result of implementing the management hardware based on our method on an FPGA, the new feature has been realized with an 2.6% increase in the number of LUTs compared to the previous architecture.

II. Full hardware implementation of RTOS-based systems

A. Concept

Fig. 1 shows the concept of the full hardware scheme Oosako proposed in [8]. The left-hand side is normal software implementation of a real-time system, where tasks (task$i$) are

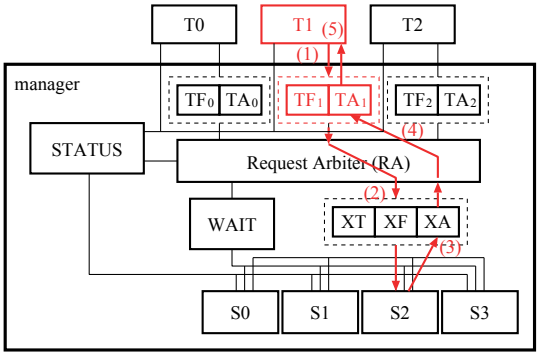


Fig. 2: Ando and Muguruma's architecture [12]

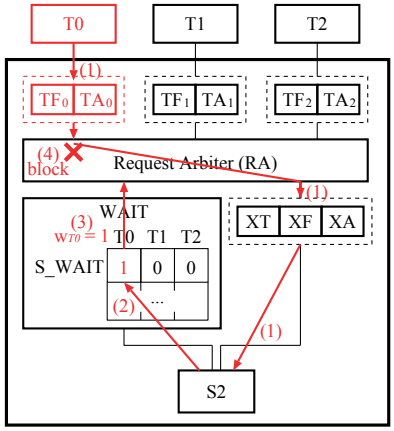


Fig. 3: Mechanism of service waiting

software programs that run on a CPU under the control of an RTOS. The right-hand side is the full hardware implementation of this system which is functionally equivalent to a CPU that runs the programs. Each of the tasks is synthesized into an independent hardware module Ti. *Manager* is a hardware module to provide function of the RTOS; it generates control signals to run/stop each task based on the state of the task, and provides services such as synchronization and communication among tasks.

Although the number of tasks handled by this scheme is limited to about 16, the response performance will be significantly improved, because the tasks are executed by hardware, there is no CPU wait, and there is no overhead caused by task scheduling nor context switching.

B. Hardware architecture

Ando and Muguruma's architecture for the full hardware scheme [12] is shown in Fig. 2. T0 through T2 are task modules. S0 through S3 are service modules that provide RTOS services such as a mutex, an eventflag, a dataqueue, etc. To avoid interference among the services, only one service is executed at a time. When multiple services are requested at the same time, the Request Arbiter (RA) mediates the requests based on the priorities of the tasks. The WAIT module keeps track of which tasks are waiting for which services. All tasks in the ready state are executed in parallel. The manager generates signals to control run/stop of each task based on the status of the task which is stored in the STATUS register.

When task T1 requests service i (mutex lock acquisition, for example) to service module S2 (the mutex module), the request is processed as follows.

(1) Task T1 requests a service by writing the ID and arguments of the service i in TF_1 and TA_1 registers, and waits for the completion of the service. The execution of T1 is suspended until the service is completed.

(2) The RA writes the ID of the highest priority task (1, in this case) into XT register, and copies TF_1 and TA_1 to XF and XA registers, respectively, by which S2 is activated.

(3) S2 executes the requested operation and writes the result value back to XA.

(4) The RA transfers the result value back to TA_1 to inform T1 of the completion of the service.

(5) T1 reads the return value from TA_1 and resumes execution.

C. Waiting and its release

1) Waiting: Some RTOS services make a requesting task wait when it cannot immediately perform the requested service. For example, when a task requests a mutex that has already been acquired by another task, or when a task tries to receive data from an empty queue, the task is forced to wait until the mutex is released or the queue receives data.

In common software implementation, the waiting requests are kept in a linear list which may be sorted by the priority or arrival order. In contrast, in the hardware architecture of [12], which task is waiting for which service is kept track of with an array of flag registers. The highest priority request is chosen by making use of the mediation function of the RA.

Fig. 3 shows how the management hardware make tasks wait. Suppose task T0 requests service i to service module S2, but it can not be processed immediately.

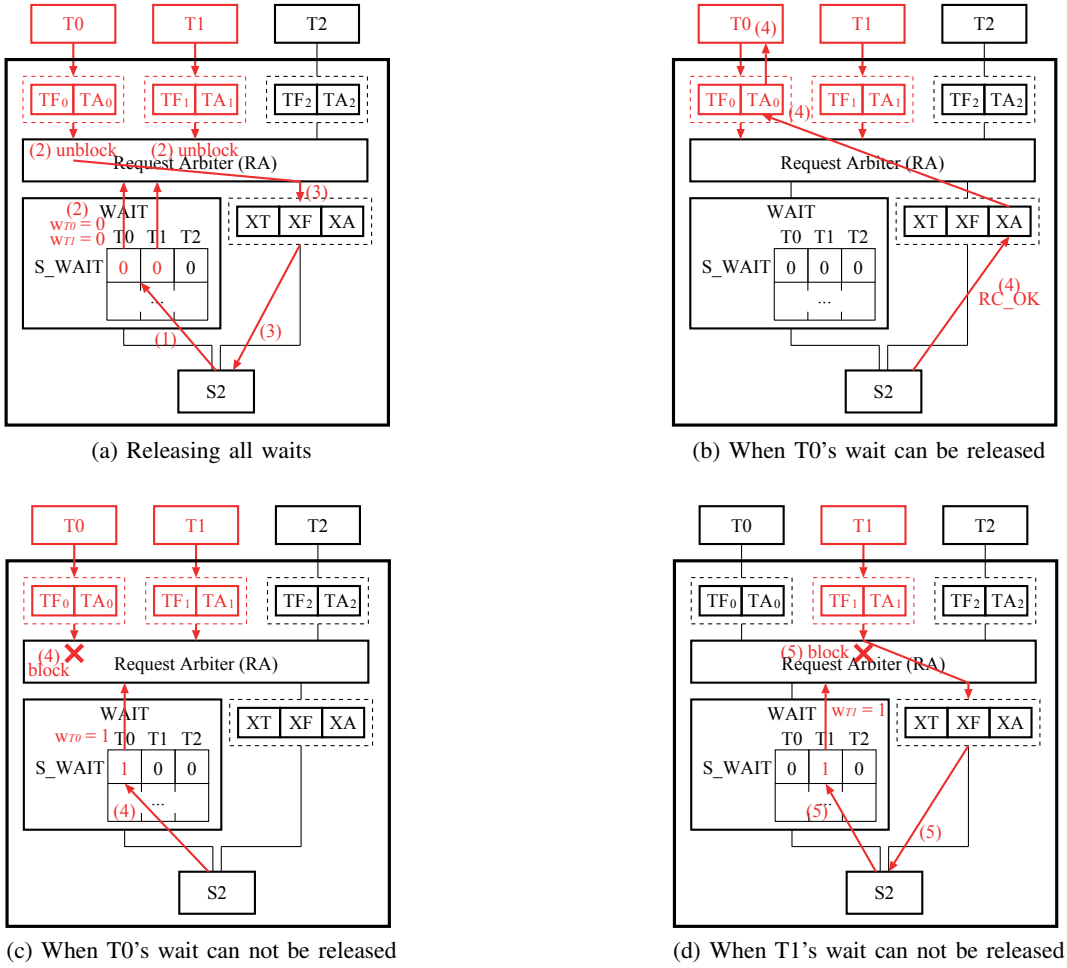
(1) T0 makes a request, which is forwarded to S2, where it turns out the request can not be executed immediately.

(2) S2 sets the flag register S_WAIT[T0][i] in the WAIT module to record T0 is waiting for service i. S2 also changes the state of T0 to the waiting state by writing to the STATUS register (which is omitted in this figure).

(3) The WAIT module emits signal $w_{T0} = 1$ to the RA to notice that T0 is waiting for some service, where the value of w_{T0} is the logical OR of S_WAIT[T0][$*$].

(4) The RA is designed to block any request from task t with $w_t = 1$ regardless of the priority of the task (i.e., it does not forward the request to XT, XF, XA). This means that the request from T0 is kept waiting without being processed.

2) Release of waiting: When the cause of the wait is resolved, the task may be released from waiting (or the request is dequeued). There are cases where multiple requests are waiting for the same service. In the architecture of [12], requests are dequeued in the order of the tasks' priorities.



(a) Releasing all waits

(b) When T0's wait can be released

(c) When T0's wait can not be released

(d) When T1's wait can not be released

Fig. 4: Mechanism of releasing waits

This priority-based processing is realized by making most of the priority-based arbitration of the RA. The flow of dequeueing is shown in Fig. 4. Here, tasks T0 and T1 are waiting for service i provided by S2 as shown in (a). The priority of each task is assumed to be higher in the order of T0, T1, and T2.

(1) S2 once releases the wait of all tasks that are waiting for service i. This is done by clearing all S_WAIT[$*$][i] at once.

(2) By (1), the WAIT module comes to send $w_{T0} = 0$ and $w_{T1} = 0$ to the RA. Then block of requests from T0 and T1 is lifted.

(3) Since the RA is designed to forward the request from the highest priority task to the service module, T0 is selected and its request is written to XT, XF, and XA, which is passed to S2.

(4) If S2 can process the request of T0 (e.g., T0 can acquire the mutex), S2 does that and notifies the completion by writing the return value of normal completion (RC_OK) to XA as shown in Fig. 4 (b). On the other hand, if S2 still can not process the request (e.g., the flags in the eventflag do not match the condition), S2 again sets S_WAIT[T0][i] to make the request wait, as in Fig. 4 (c).

(5) Regardless of whether the T0's wait is released or not, the next request (from task T1) is conveyed to S2 as shown in Fig. 4 (d). If S2 cannot process the request (i.e., the mutex is locked by T0), it sets S_WAIT[T1][i] again to make the request wait.

One reason for releasing all waiting request once is to let the RA select the highest priority task. The other reason is that there are services such as the eventflag that can release wait of multiple tasks whose requests match the condition.

The problem with this design is that only priority-based dequeueing is possible. There are some RTOSs or some particular services that assume arrival order dequeueing. For example, TOPPERS/ASP3's message buffer performs arrival order dequeueing, while mutex and eventflag can select between arrival order or priority order for each instance.

Moreover, most of the RTOSs stipulates that requests are processed in their arrival order when they are from the tasks of the same priority. This cannot be realized by this architecture.

In addition, since the RA operates by looking up only task's priorities, a request from the highest priority task may be processed even if the request is irrelevant from the wait, which prolongs the process of the wait release.

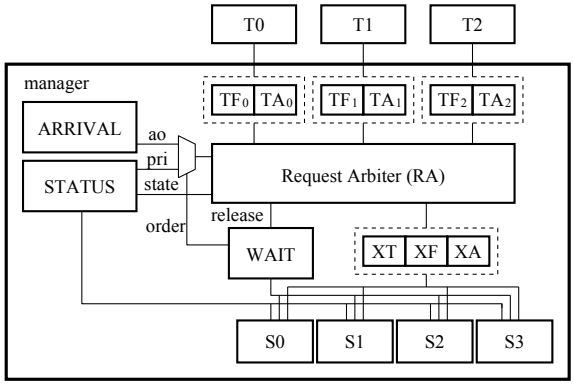


Fig. 5: Proposed architecture

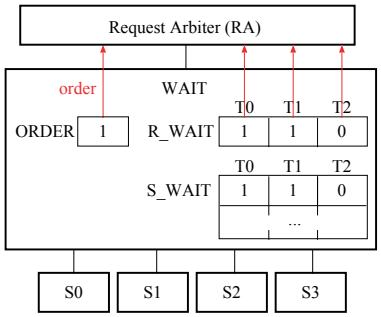


Fig. 6: WAIT module

III. Arrival order processing

A. Overview

In this paper, we propose a hardware scheme that enables both arrival order processing and priority order processing of service requests in the architecture of [12]. The proposed design allows each instance of the service may choose between the two orders.

In order to prevent wait release from being delayed, wait release and normal mediation are clearly separated at the RA. For this purpose, the RA is redesigned to have two modes, one to complete wait release and the other to mediate normal service requests. In the wait release mode, the RA is instructed to dequeue requests in the priority order or in the arrival order.

We have also designed a module to record the arrival order of requests. Instead of logging the timestamps of requests, only the arrival orders are maintained so as to reduce hardware cost.

The overall configuration the proposed design is shown in Fig. 5. A module for arrival order recording (ARRIVAL) has been added to the architecture of [12]. The WAIT module emits an *order* signal by which the priority fed to the the RA is switched. It also sends *release* signal to the RA, which place the RA into the release mode.

B. WAIT module and RA

Fig. 6 shows the structure of the revised WAIT module. It has a one-dimensional flag R_WAIT and a single bit register ORDER, in addition to the two dimensional flag register S_WAIT (to record which task is waiting for which service). R_WAIT[t] is set when task t is being released from waiting, and ORDER indicates in which order the waits should be released (e.g., 0 for the priority order and 1 for the arrival order).

When a service module wants to dequeue requests for service i, it sets R_WAIT[t] for all t where S_WAIT[t][i] = 1 (instead of clearing all S_WAIT[$*$][i]). When any of R_WAIT[$*$] is set to 1, release = 1 is send to the RA to switch it to the wait release mode. The RA in the wait release mode unblocks the request from the task which has the highest order among the tasks whose R_WAIT[$*$] is 1. This is repeated while any of R_WAIT[$*$] is 1, and then the RA returns to the normal mode.

The following is the flow when the service module S releases the wait of the requests for service i.

(1) Service module S instructs WAIT to release waiting tasks by specifying instance i and the order (priority/arrival) of release.

(2) WAIT sets R_WAIT[t] for every task t with S_WAIT[t][i] = 1. It also sets ORDER register as instructed by the service module S. WAIT sends the value of R_WAIT to the RA to place it into the wait release mode.

(3) The RA in the wait release mode selects task t with the highest order among the tasks with R_WAIT[*] = 1, and pass the request from t to the service module via XT, XF, XA.

(4) If service module can process the request from t, it clears S_WAIT[t][i] and R_WAIT[t] to indicate that the wait is released. If not, only R_WAIT[t] is cleared, which causes the request of task t to be blocked again by the RA.

The number of tasks to be dequeued depends on the service. Most of the services, such as the mutex and the eventflag with the CLR attribute, dequeue only one waiting request. In this case, the service module clears all R_WAIT[$*$] at the end of the release, which brings the RA back to the normal mode. On the other hand, in the case of the eventflag without the CLR attribute, the dequeue operation is continued until all requests are tested.

C. Switching between priority order and arrival order

The order for task wait release is instructed to the RA by changing the priority inputs to the RA. The circuit for this scheme is shown in Fig. 7, where pri_t and ao_t are the current priority and the arrival order of task t, respectively, and MSB_t and LSB_t are the upper and the lower bits of the priority for task t, respectively. When an instructed order is "priority," the outputs are set as $\text{MSB}_t = \text{pri}_t$ and $\text{LSB}_t = \text{ao}_t$, and vice versa when an order is "arrival." This allows the RA to arbitrate tasks of the same priority in the arrival order when the priority order is instructed.

D. Arrival order recording module

In the arrival order recording module, only the orders in which the services are requested are recorded, because

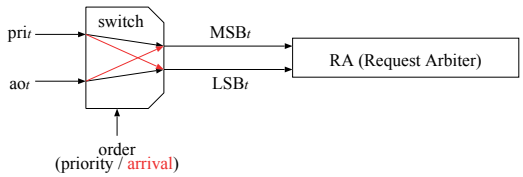


Fig. 7: Switching between priority order and arrival order

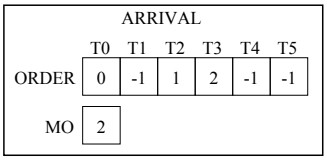


Fig. 8: Arrival order recording module

recording of timestamps would result in many bits of registers and a large comparator tree.

In our design, when k tasks are waiting for the completion of a service, their orders are recorded by consecutive integers $0, 1, 2, \ldots, k-1$. The order of a task that are not in the waiting state is represented by -1. The orders are updated each time tasks request services or a service module completes a requested service.

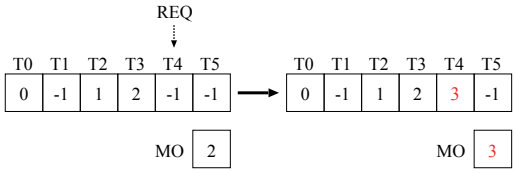
The structure of the ARRIVAL module is shown in Fig. 8. Register ORDER is a one-dimensional array to record the arrival orders of the tasks. Register MO stores the maximum value of the arrival order. In this example, 3 tasks are in the waiting state where T0, T2, and T3 have issued requests in this order.

When task t newly requests a service, MO is incremented and the new value of MO is assigned to ORDER[t]. When the service requested by task t' is completed, ORDER[t] for all t satisfying ORDER[t']<ORDER[t] is decremented, and then MO is decremented and ORDER[t'] is reset to -1.

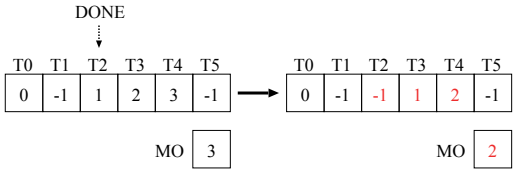
For example, in Fig. 9 (a), T4 requests a service, then MO is incremented to 3 and the new value is set to ORDER[T4]. When the service for T2 is completed, as shown in Fig. 9 (b), ORDER[T3] and ORDER[T4] are decremented because they were higher ORDER[T2], and then MO is decremented and ORDER[T2] is reset to -1,

When multiple tasks request services at the same time, the number of requests is added to MO, and the ORDER is recorded in the order of the task ID. For example, as shown in Fig. 9 (c), when T1 and T5 request services at the same time, MO is double incremented to 4 and ORDER[T1] and ORDER[T5] are updated to 3 and 4, respectively. This operation is implemented by adding up the request signals from the tasks whose ID is smaller than each task by a parallel counter.

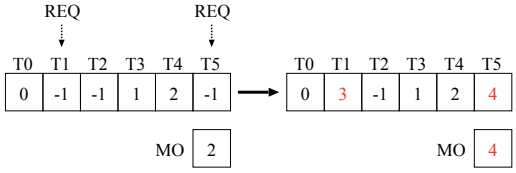
If a request is issued at the same time as service completion, all the ORDERs higher than the task to be completed are decremented, and the ORDER of the requesting task is set to MO. Fig. 9 (d) shows an example. When T2 requests a service



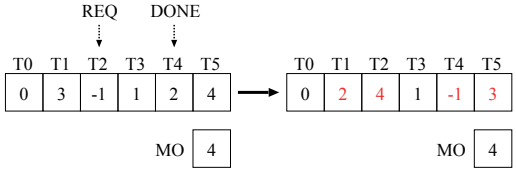
(a) Receipt of a service request



(b) Completion of a service request



(c) Receipt of simultaneous requests



(d) Simultaneous receipt and completion of requests

Fig. 9: Recording of arrival order

and T4 completes a service at the same time, ORDER[T1] and ORDER[T5] which are higher than ORDER[T4] are decremented and the value of MO is set ORDER[T2].

IV. Implementation and Experiment

Based on the proposed scheme, a management hardware assuming TOPPERS/ASP3 has been designed by Verilog HDL. The hardware description has been synthesized by Xilinx Vivado 2020.2 targeting Xilinx FPGA Artix-7. The details of the service modules in the manager are as follows:

- The control_task module provides the following 12 services:
  act_tsk, can_act, ter_tsk, chg_pri, get_pri, wup_tsk, can_wup, rel_wai, sus_tsk, rsm_tsk, loc_cpu, unl_cpu
- The shared_variable module has 32 words each consisting of 32bits.
- The mutex module implements 2 mutexes.
- The eventflag module has 2 registers each consisting of 32 bits.
- The dataqueue module has 2 buffers consisting of 10 words.

The circuit size of each module is shown in TABLE I. #LUT is the number of look-up tables and #FF is the number of flip-flops. “task $\times$ 4” is the total circuit size of the 4 tasks. Since

TABLE I: Circuit size

module	previous [12]		proposed	
	#LUT	#FF	#LUT	#FF
task × 4	48	44	48	44
control_task	1053	140	1047	140
shared_variable	125	0	125	0
mutex	1881	48	1956	48
eventflag	151	64	151	64
dataqueue	1108	16	1081	16
manager	898	2066	992	2089
total	5264	2378	5400	2401

TABLE II: Critical path delay

	previous [12]	proposed
delay[ns]	7.871	7.803

the hardware description was the same for both the previous and proposed designs, the circuit sizes are the same.

Among the 5 service modules, the control_task and shared_variable modules are irrelevant to the processing order (hence, their hardware descriptions were the same for the both designs). On the other hand, the other 4 modules are added with wires necessary for arrival order processing. Although there were some fluctuations, the total circuit size increased only by 1.0% in terms of the number of LUTs compared to the previous architecture.

The increase in the size of manager's circuits was 10.5% in terms of the LUT count. We guess this increase is due to the addition of circuits such the arrival order recording module.

In total, the numbers of LUTs and FFs have increased by 2.6% and 1.0%, respectively, which we believe are small enough.

As shown in TABLE II, there has been no increase in critical path delay of the synthesized circuit

V. Conclusion

In this paper, a new hardware scheme for dealing with arrival order processing as well as priority order processing of service requests in full hardware implementation of RTOS-based systems. The new ideas are the mechanism to switch between the two orders, the wait release mode of the request arbiter, and the arrival order recording module.

Synthesis result of an example circuit has shown that the new feature was implemented with only 2.6% increase in the circuit size and there was no increase in the critical path delay.

The manager module of the experiment was designed manually. We are now working on automatic generation of the manager module [14]. Application to design of practical real-time systems is another future work.

Acknowledgment

Authors would like to express their appreciation to Dr. Hiroyuki Kanbara of ASTEM/RI, Prof. Hiroyuki Tomiyama of Ritsumeikan University, and Mr. Takayuki Nakatani (formerly with Ritsumeikan University) for their discussion and valuable comments. We would also like to thank to the members of Ishiura Lab. of Kwansei Gakuin University. This work was partly supported by JSPS KAKENHI under Grant No. 19H04081.

References

[1] Y.-C. Cho, S.-J. Yoo, K.-Y. Choi, N.-E. Zergainoh, and A. Jerraya: "Scheduler implementation in MPSoC design," in *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC 2005)*, pp. 151–156 (Jan. 2005).

[2] M. Vetromille, L. Ost, C. A. M. Marcon, C. Reif, and F. Hessel: "RTOS scheduler implementation in hardware and software for real time applications," in *Proc. International Workshop on Rapid System Prototyping (RSP '06)*, pp. 163–168 (June 2006).

[3] P. Kohout, B. Ganesh, and B. Jacob: "Hardware support for real-time operating systems," in *Proc. International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '03)*, pp. 45–51 (Oct. 2003).

[4] T. Nakano, Y. Komatsudaira, A. Shiomi, and M. Imai: "Performance evaluation of STRON: A hardware implementation of a real-time OS," in *IEICE Trans. Fundamentals*, vol. E82-A, no. 11, pp. 2375–2382 (Nov. 1999).

[5] N. Maruyama, T. Ishihara, and H. Yasuura: "An RTOS in hardware for energy efficient software-based TCP/IP processing," in *Proc. IEEE Symposium on Application Specific Processors (SASP 2010)*, pp. 58–63 (June 2010).

[6] S. Shibata, S. Honda, H. Tomiyama, and H. Takada: "Advanced system-builder: A tool set for multiprocessor design space exploration," in *Proc. International SoC Design Conference (ISOCC 2010)*, pp. 79–82 (Nov. 2010).

[7] Y. Ando, S. Honda, H. Takada, M. Edahiro: "System-level design method for control systems with hardware-implemented interrupt handler," *IPSJ Journal of Information Processing*, vol. 23, no. 5, pp. 532–541 (Sept. 2015).

[8] Y. Oosako, N. Ishiura, H. Tomiyama, and H. Kanbara: "Synthesis of full hardware implementation of RTOS-based systems," in *Proc. International Symposium on Rapid System Prototyping (RSP 2018)*, pp. 1–7 (Oct. 2018).

[9] TOPPERS/ASP kernel, https://www.toppers.jp/ (accessed 2023-01-07).

[10] FreeRTOS kernel, https://www.freertos.org/ (accessed 2023-01-07).

[11] W. Nakano, Y. Shinohara, and N. Ishiura: "Full Hardware Implementation of FreeRTOS-Based Real-Time Systems," in *Proc. IEEE Region 10 Conference* (Dec. 2021).

[12] T. Ando, I. Muguruma, Y. Ishii, N. Ishiura, H. Tomiyama, and H. Kanbara: "Full Hardware Implementation of RTOS-Based Systems Using General High-Level Synthesizer," in *Proc. Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2022)*, pp. 2–7 (Oct. 2022).

[13] H. Minamiguchi, M. Nakahara, Y. Ishii, Y. Shinohara, I. Muguruma, and N. Ishiura: "Hardware RTOS Services for Full Hardware Implementation of RTOS-Based Systems," in *Proc. Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2022)*, pp. 14–19 (Oct. 2022).

[14] H. Minamiguchi, N. Ishiura, H. Tomiyama, and H. Kanbara: "Automatic Generation of Management Module for Full Hardware Implementation of RTOS-Based Systems," in *Proc. International Technical Conference on Circuit/Systems, Computers and Communications (ITC-CSCC 2023)* (June 2023).