Full Hardware Implementation of RTOS-Based Systems Using General High-Level Synthesizer

Takuya ANDO †,* Iori MUGURUMA †,** Yugo ISHII Nagisa ISHIURA
Kwansei Gakuin University
1 Gakuen Uegahara, Sanda, Hyogo, 669-1330 Japan

Hiroyuki TOMIYAMA
Ritsumeikan University
1-1-1 Noji–Higashi, Kusatsu, Shiga, 525-8577 Japan

Hiroyuki KANBARA
ASTEM RI/KYOTO
134 Chudoji Minamimachi, Shimogyo-ku, Kyoto, 600-8813 Japan

Abstract—This article proposes a method for implementing an RTOS-based system as hardware using a general high-level synthesizer. Oosako proposed a full hardware scheme where all the tasks and all the RTOS functions are implemented as hardware. However, it depended on special features of an in-house binary synthesizer ACAP; a synthesized hardware module has a stall port by which module's execution can be suspended, and accesses to global variables are automatically translated to accesses to the single memory space without rewriting the source program. Moreover, the size of the resulting circuits was too large for practical use. This paper proposes a new architecture that can dispense with the stall ports and also reduces the size of the resulting circuits. This paper also presents a wrapper class for global variable accesses and a style of programs to minimize the rewriting of task programs. Based on the proposed method, a hardware module for a reduced version of "sample1" bundled with TOPPERS/ASP3 has been successfully implemented as hardware using Xilinx Vitis HLS. Moreover, the size of the resulting circuit was 89% smaller than that by the previous method.

I. Introduction

Owing to the recent advances in information and network technologies, various new digital devices and services are being deployed in our everyday life. Accordingly functionalities implemented in embedded devices are getting more and more rich and sophisticated. In addition to this, in some areas such as an unmanned aerial vehicle, autonomous cars, and service robots, high response performance is also required.

Such real-time systems, where tasks must be processed within the specified periods in response to input events, are implemented using a real-time operating system (RTOS). The RTOS helps designers to implement real-time systems by providing controllability and predictability of execution time of concurrent tasks. However, it is getting more and more difficult to ensure real-time performance as the complexity of the systems grows.

One of the solutions to this problem is hardware acceleration. There have been many efforts to implement some or all functions of RTOSes as hardware [1, 2, 3, 4, 5]. On the other hand, there have also been some attempts to convert software tasks into hardware [6, 7] using high-level synthesis [8]. While these methods implement a part of systems as hardware, [9] and [10] have implemented a whole system as hardware, which however only dealt with bare metal systems.

Oosako has proposed a full hardware implementation of RTOS-based systems [11], where both tasks and RTOS functions are synthesized into hardware by high-level synthesis. Assuming that the tasks are created statically at compile time, every task is synthesized into an independent hardware component which can run in parallel with the other tasks. The scheduler is reduced to a simple controller without ready queues. Small sample programs based on TOPPERS/ASP3[1] [11] and FreeRTOS[2] [12] were successfully implemented as hardware.

Although this scheme drastically enhanced response performance of systems, there are two issues to be resolved for its practical use. One is that the scheme depends on an in-house binary synthesizer ACAP (Assembly Compatible Architecture Prototyper) [13]. This is because ACAP generates hardware modules with the stall ports which are convenient for controlling task execution and because ACAP can handle global shared variables in a natural way. Dependence on this specific synthesizer heavily limits the applicability of the scheme. The other issue is that resulting hardware is too large. This is partly due to the performance of the synthesizer but also due to the architecture where multiple copies of hardware for RTOS services are present across tasks.

This paper proposes a new architecture and a new synthesis method for full hardware implementation of RTOS-based systems using a general high-level synthesizer. The architecture of the hardware is revised so that duplication of hardware is eliminated by collecting hardware for services in one place. In order to make it possible to generate hardware by a general high-level synthesizer, a new mechanism for task control which eliminates the need for the stall ports is proposed. A wrapper class which makes the handling of global shared variables easier is also proposed.

Based on the proposed method, a sample program "sample1" bundled with TOPPERS/ASP3 has been successfully

*Currently with ROHM Co., Ltd., Japan.
**Currently with Honda Motor Co., Ltd., Japan.

[1] https://www.toppers.jp/
[2] https://www.freertos.org/

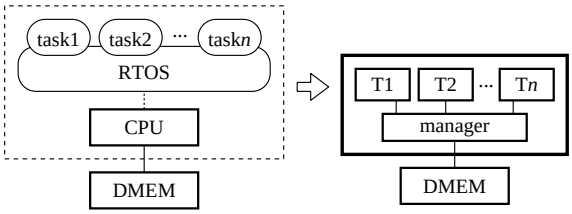

Fig. 1. Full hardware implementation of RTOS-based system

synthesized into full hardware by a commercial high-level synthesizer Xilinx Vitis HLS. The resulting circuit size was 89% smaller than that by the previous method.

II. Full hardware implementation of RTOS-based systems

A. Full hardware implementation

An RTOS (Real-Time Operating System) runs multiple sequential programs, called tasks, concurrently. It controls, or schedules, execution of the tasks based on their priorities so that required work associated with input events to the system will be performed within the specified periods.

In [11], a method is proposed that synthesizes a given application program written using RTOS system calls into a hardware module which is functionally equivalent to a CPU that runs the program, as shown in Fig. 1. "task1" through "task*n*" are software tasks running under an RTOS, which are converted into hardware modules T1 through T*n* by high-level synthesis [8]. "Manager" is a dedicated hardware module that works in place of the RTOS; it controls execution of the tasks and provides services such as mutexes and data queues.

In this hardware scheme, all the task modules are executed in parallel as soon as they become ready. The manager controls the task modules by *stall* signals. A task module stops when the stall signal to the task is 1 and runs otherwise. The stall signal to a task module is generated from the task's state kept in the task status register.

Response time of the system will be drastically reduced by this scheme due to 1) parallel execution of tasks (no CPU wait), 2) no overhead regarding scheduling nor context switching, and 3) hardware acceleration.

On the other hand, there are limitations to this scheme. First of all, all the tasks must be statically created (at compile time). The manager can not handle many tasks so that the number of tasks should be at most 16. In order to avoid interference among RTOS services, service calls are processed one by one (sequentially), in spite that the tasks are executed in parallel.

B. Oosako's architecture

Fig. 2 illustrates the architecture of resulting hardware in the Oosako's scheme [11].

T1, T2, and T3 are hardware modules for tasks, which are henceforth called task modules. The manager has a status register for each task, which contains the state, the priority, the timer, etc. regarding the task. The manager controls execution of the task modules by the stall signals; a task module runs normally when the stall signal is 0, and holds its execution when

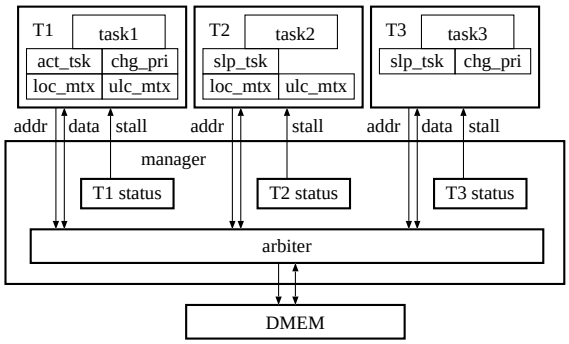

Fig. 2. Architecture of resulting hardware in Oosako's scheme [11]

```
 1: ER act_tsk(ID tskid) {
 2:
 3:   if (IS_TASK_CONTEXT && tskid == TSK_SELF) {
 4:     tskid = TOPPERS_HW_SELF_ID;
 5:   }
 6:
 7:   if (tskid <= 0 || TNUM_TSKID < tskid) {return E_ID;}
 8:   volatile T_RTSK *target =
 9:     &(task_status[tskid-1].rtsk);
10:
11:   _loc_service_call();
12:
13:   uint_t actcnt = target->actcnt;
14:
15:   ER rc = E_OK;
16:   if (glob_status.f_cpu_locked) {rc = E_CTX;}
17:   else if (actcnt >= TMAX_ACTCNT) {rc = E_QOVR;}
18:   else if (target->tskstat != TTS_DMT) {rc = E_QOVR;}
19:   else {target->tskstat = TTS_RDY;}
20:
21:   _unl_service_call();
22:
23:   return rc;
24: }
```

Fig. 3. Rewritten code of act_tsk

the stall signal is 1. The stall signal of a task module is generated from the status register of the task; the stall signal is set to 0 if the task status is in the *Running* state, and set to 1 otherwise. Note that when a task become *Ready* then the manager forces the state of the task to *Running* in the next clock cycle.

The task modules read and write DMEM (data memory) through the manager. Simultaneous accesses to DMEM are sequentialized by the arbiter. The task status registers are mapped in the address space, so that task modules may access those registers by load/store operations.

A task module consists of the hardware for the task itself and the hardware to execute RTOS services which the task calls. In this figure, task1 is assumed to call four services (act_tsk, chg_pri, loc_mtx, and unl_mtx). They are synthesized into a single task module by high-level synthesis.

Since the status registers can be accessed as variables, the original source code of the service calls in the RTOS can be used as inputs to high-level synthesis with a little modification. For example, a service call act_tsk of TOPPERS/ASP3, which changes the state of a specified task to *Ready*, can be rewritten as shown in Fig. 3. In line 19, it assigns `TTS_RDY`, the value for *Ready*, to the task status register. Most of the other lines are for error handling. Lines 11 and 21 are lock and unlock to avoid parallel execution of other service calls.

Oosako's scheme assumes the use of a specific synthesizer ACAP [13]. It is a binary synthesizer which generates hardware from MIPS R3000 binary codes. It can be also used as a high-level synthesizer by feeding binary codes generated by GCC from C or C++ programs. ACAP is used for two reasons. ACAP generates an explicit *stall* port for each hardware module. The state transition of the module is disabled via this port at any time, which the manager hardware can make full use of to control tasks. The other reason is that ACAP is convenient to handle shared (global) variables. ACAP maps all the global variables in the memory space and enables task modules to access the variables via address and data ports.

C. Circuit size and universality issues

Although the full hardware scheme enables implementation of extremely fast real-time systems, there are two issues to be resolved; the huge circuit size and dependence on the specific synthesizer.

In the preliminary implementation in [11], the resulting circuit from a simple sample bundled with TOPPERS/ASP3 consumed 40,000 LUTs (Xilinx Artix-7), which is impractically large. One reason for this is the low performance of ACAP, but there is an architectural issue; each task owns hardware for necessary RTOS services, which results in multiple copies of the same hardware that are never executed in parallel.

Moreover, the dependence on ACAP is a hurdle for broad use. Since user codes and the body of RTOS services are usually written in C or C++, there should be no need for binary synthesis. However, not all the high-level synthesizer generate hardware modules with the stall ports, and there is no scheme to describe the stall-at-anytime behavior in C or C++. How to write accesses to global shared variables is also an issue. Though general high-level synthesizers may generate hardware modules with address and data ports to access external variables, this involves the need for rewriting of the user codes, which is troublesome if there are many global variables.

III. Full hardware implementation by general synthesizer

A. Overview

This paper proposes a new architecture and a new synthesis scheme to realize full hardware implementation of RTOS-based systems using a general high-level synthesizer. The technical highlights are as follows:

1. Duplication of hardware to provide RTOS services is eliminated by moving the service hardware from the tasks to the manager.
2. Execution of task hardware modules is controlled via usual read ports instead of the stall ports. The pause of the tasks is realized as wait for service completion.
3. User code rewriting for shared variable accesses is minimized by defining a wrapper class for the global variable access and functions-inside-functions.

```
{
  ...
  chg_pri(TSK1, LOW_PRI);
  ...
}
```

(a) Service call from a task

```
#define chg_pri(tskid, tskpri) \
       _chg_pri(tskid, tskpri, _F, _A0, _A1)

ER _chg_pri(ID tskid,
            PRI tskpri,
            volatile int* const _F,
            volatile int* const _A0,
            volatile int* const _A1){
  *_A0 = tskid;
  *_A1 = tskpri;
  ap_wait(); // synchronization
  *_F = SERV_CTRL_TSK | METHOD_CHG_PRI;
  ap_wait(); // wait for the result
  return *_A0;
}
```

(b) Body of service call (stub)

Fig. 4. C codes regarding service call

B. Centralized service

Hardware to provide RTOS services is moved from the tasks to the manager. Fig. 5 (a) and (b) shows the previous and the new architectures, respectively. In the new architecture, the RTOS services are executed by the manager not by the task modules. This eliminates duplication of hardware.

A task requests a service to the manager by writing the ID of the service and necessary arguments into the control registers F and A for the task. The manager passes the request to the service module in charge of the request. When there are multiple requests at the same time, the request arbiter (RA) selects one of them according to the priorities of the tasks that issued the requests. The other requests wait for their turn. The called service module processes the request by accessing status registers and memories if necessary, and returns the results to the manager, which are passed back to the task via the register A.

Although the (new) service modules could be generated by high-level synthesis, most of them can be manually designed in RTL since they usually result in simple circuits with a few or several states.

Note that the access to shared variables is now dealt with as one of the services. This is partly for unifying the arbiter for memory access to the arbiter for services, but also for a new task control scheme for eliminating the stall ports, as described in the next subsection.

An example of C codes regarding a service call in the new scheme is shown in Fig. 4. (a) is a caller code (a user code), which needs no modification. (b) is the body of `chg_pri`, which writes two arguments and the service ID to `*_A0`, `*_A1` and `*_F` that are port variables connected to the control registers. It waits for the return code to be written into `*_A0` by the manager. `ap_wait()` is a synthesizer specific macro that instructs that subsequent statements must not be scheduled before the preceding statements. The stub function (`_chg_pri`, in this case) may be inline expanded into the task body, which becomes an input to high-level synthesis.

C. Task control scheme

In order to realize the task control without using stall ports, a new control policy is introduced. That is:

- Tasks are allowed to run even when they are not in the *Running* state.
- However, only the service requests from running tasks are processed. Thus, when a non-running task makes a service call, it waits until it becomes *Running* and the service is processed.

Non-running tasks are allowed to run to update their local states but could not affect the other tasks nor the output of the system because all the system calls and accesses to the shared variables are blocked. This realizes equivalent system semantics as the one with the stall ports.

Fig. 5 (c) shows how this scheme is implemented in hardware. Instead of sending the stall signals to the task modules, they are used to block the service requests so that requests from non-running tasks can not participate in the arbitration until they become *Running* again.

D. Wrapper class for shared variable accesses

Accesses to shared variables are dealt with in the same way as service calls. Fig. 6 shows stub functions `M_READ_int` and `M_WRITE_int` to request read and write integer data, respectively.

Given a task code (user code) as shown in Fig. 7 (a), it must be converted to a code in (b) to be an input to high-level synthesis. Since this rewriting across all the tasks is a burden for users, a wrapper class to reduce rewriting is defined. As shown in (c), users only have to declare the variables as of the shared type (`G_int`). The wrapper class can be defined as in Fig. 8.

There remains a technical issue in translating user codes to synthesizable codes. Generally, a task is composed of multiple functions, among which the variables to access control registers and the class objects of shared variables must be shared. We make this possible by converting task functions into functions-inside-functions in a main function.

For example, suppose a task program shown in Fig. 9 (a) is given. It declares global (shared) variable `x` and `y`, and two functions `sub` and `tsk`, where `tsk` is the main function. This code is converted into (b). A new function `tsk_main` is a wrapper function which is synthesized into a task module. It has three arguments, which will result in the external ports. Global variables `x` and `y` are converted into instances of the wrapper class. Functions `sub` and `tsk` are defined as functions-inside-functions, from which `_F`, `_A0`, `_A1`, `x`, and `y` can be accessed like global variables. Note that no change is needed for the bodies of `sub` and `tsk` so this conversion may be done automatically.

IV. Experimental Result

Based on the proposed method, a sample program "sample1" bundled with TOPPERS/ASP3 has been implemented as hardware. This program is composed of a main

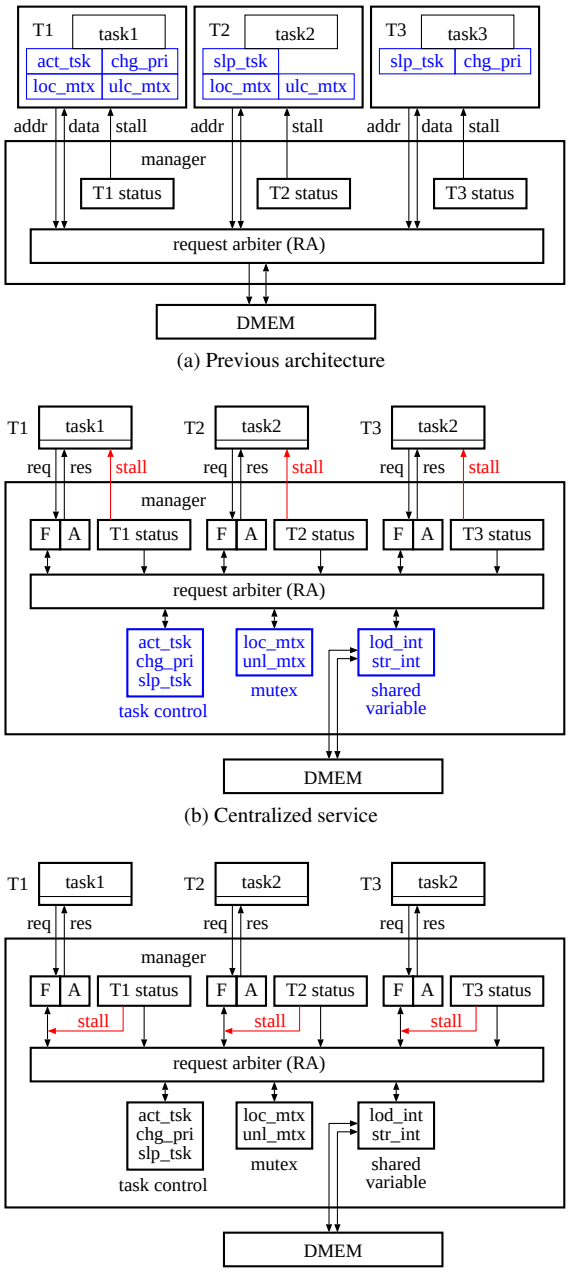

Fig. 5. Proposed architecture

task MAIN_TASK, a task to handle exception EXC_TASK, and three concurrent tasks TASK1, TASK2, and TASK3. MAIN_TASK receives a message from a serial port and calls the following 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

At this point, the alarm, cyclic and interrupt handlers have not been implemented yet, and the service calls related to the handlers were deleted.

The service modules and the managers are manually designed in Verilog HDL and logic synthesized by Xilinx Vivado (2020.2) targeting Xilinx FPGA Artix-7 (xc7a100tcsg324-3).

```
ER M_READ_int (int addr,
               volatile int* const _F,
               volatile int* const _A0){
  *_A0 = addr;
  ap_wait();
  *_F = SERV_GRW | METHOD_READ;
  ap_wait();
  return *_A0; // read data
}

ER M_WRITE_int (int addr,
                int value,
                volatile int* const _F,
                volatile int* const _A0,
                volatile int* const _A1){
  *_A0 = addr;
  *_A1 = value;
  ap_wait();
  *_F = SERV_GRW | METHOD_WRITE;
  ap_wait();
  return *_A0; // notification of completion
}
```

Fig. 6. Stub functions for shared variable accesses

```
int x;
int y;

{
  ...
  x = 1;
  y = x + 2;
  ...
}
```

(a) Original task code

```
#define X_ADDRESS 0x80000000
#define Y_ADDRESS 0x80000004

{
  ...
  M_WRITE_int(X_ADDRESS, 1);
  M_WRITE_int(Y_ADDRESS, M_READ_int(X_ADDRESS) + 2);
  ...
}
```

(b) Modified code

```
{
  G_int x(_F, _A0, _A1);
  G_int y(_F, _A0, _A1);
  ...
  x = 1;
  y = x + 2;
  ...
}
```

(c) Code using wrapper class

Fig. 7. Task codes for shared variable accesses

Task modules are synthesized by ACAP (2016.10) [13] for the methods in [11] and Xilinx Vitis HLS (2020.2) for the proposed method.

The size of the synthesized circuit is listed in Table I (a), where "#LUT" and "#FF" are the numbers of LUTs and flipflops, respectively. The size of the manually designed modules ("top" through "manager") are about the same. On the other hand, the size of the task modules is drastically reduced by the proposed method. This is partly due to the elimination of duplicated service hardware but mainly owing to the performance of the high-level synthesizer.

Table I (b) shows the critical path delay of the circuits, which is also reduced by the proposed method.

```
static int addr = 0x80000000;

class G_int{
  const int address;
  volatile int* const _F;
  volatile int* const _A0;
  volatile int* const _A1;

public:
  G_int(
    volatile int* const f,
    volatile int* const a0,
    volatile int* const a1
  ) : address(addr), _F(f), _A0(a0), _A1(a1) {addr += 4;}

  operator int () { return M_READ_int(address, _F, _A0); }

  G_int& operator = (int value) {
    M_WRITE_int(address, value, _F, _A0, _A1);
    return *this;
  }

  G_int& operator = (G_int& x) {
    *this = (int) x;
    return *this;
  }
};
```

Fig. 8. Wrapper class for shared variables

```
int x;
int y;

void sub(){
  x = x + 3;
  y = y + x;
}

void tsk(){
  x = 1;
  y = x + 2;
  chg_tsk(TSK1, LOW_PRI);
  sub();
}
```

(a) Original task code

```
void tsk_main(
  volatile int* const _F,
  volatile int* const _A0,
  volatile int* const _A1
  ){

  G_int x(_F, _A0, _A1);
  G_int y(_F, _A0, _A1);

  auto sub = [=]() mutable {
    x = x + 3;
    y = y + x;
  };

  auto tsk = [=]() mutable {
    x = 1;
    y = x + 2;
    chg_tsk(TSK1, LOW_PRI);
    sub();
  };

  tsk();
}
```

(b) Converted code for synthesis

Fig. 9. Conversion to synthesizable code

The response performance of the circuits are summarized in Table II. "#cycle" is the number of clock cycles each service takes after a task calls the service until the task receives the return code, and the "latency" is the product of "#cycle" and the critical path delay. All the task control related services are executed well in 150ns, which is fast enough even for extreme applications.

V. Conclusion

This paper has proposed a new method for full hardware implementation of RTOS-based systems using a general high-level synthesizer. It is based on migration of service hardware from tasks to the managers, a new task control scheme, and use of a wrapper class for shared variable accesses. The new scheme has drastically reduced the resulting circuit size.

TABLE I
RESULT OF SYNTHESIS OF `sample1`

(a) Circuit size

module	method in [11] (ACAP) #LUT	#FF	proposed method (Vitis HLS) #LUT	#FF
top	368	5	0	0
arbiter	383	5	-	-
serv_grw	–	–	300	1,025
serv_ctrl_tsk	–	–	990	146
manager	4,079	2,672	2,394	3,041
TASK1	5,520	929	103	218
TASK2	7,567	925	104	219
TASK3	7,189	965	104	219
MAIN_TASK	6,199	943	323	559
EXC_TASK	8,003	939	8	8
total	39,313	7,383	4,326	5,435

(b) Critical path delay [ns]

method in [11] (ACAP)	proposed method (Vitis HLS)
13.108	9.783

HLS: ACAP (2016.10), Xilinx Vitis HLS (2020.2)
Logic synthesizer: Xilinx Vivado (2020.2)
Target: Xilinx Artix-7 (xc7a100tcsg324-3)

TABLE II
RESPONSE PERFORMANCE OF SERVICES

service call	method in [11] #cycle	latency [ns]	proposed method #cycle	latency [ns]
act_tsk	23	301.3	10	97.8
wup_tsk	9	125.4	10	97.8
ext_tsk	12	157.2	10	97.8
ras_ter	10	139.3	9	88.1
ter_tsk	8	111.5	7	68.5
slp_tsk	16	209.6	15	146.7

We have already implemented service modules for synchronization and communication, such as mutexes, event flags, and data queues [14], and we are now working on dynamic memory allocation such as memory pools. Our first target is TOPPERS/ASP3, but we are now working on adapting our scheme to support FreeRTOS. At this point, the manager module is manually designed. We are also working on automatic generation of Verilog HDL codes for the manager for an arbitrary number of tasks and services. Using this generator, we expect that we can identify bottlenecks and improve or optimize the architecture of the manager module so that more than 16 tasks can be handled.

Acknowledgments

Authors would like to express their appreciation to Mr. Takayuki Nakatani (formerly with Ritsumeikan Univ.), Mr. Shimpei Tamura (formerly with Kwansei Gakuin Univ.) for their discussion and valuable comments. We would also like to thank to the members of Ishiura Lab. of Kwansei Gakuin Univ. This work was partly supported by JSPS KAKENHI under Grant Nos. 19H04081, 20H00590, and 21K19776.

REFERENCES

[1] Y. Cho, S. Yoo, K. Choi, N-E Zergainoh, and A. A. Jerraya: "Scheduler implementation in MPSoC design," in *Proc. 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. RSP '06* pp. 163–168 (June 2006).

[3] 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).

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

[5] P. Kohout, B. Ganesh, and B. Jacob: "Hardware support for real-time operating systems," in *Proc. CODES+ISSS '03*, pp. 45–51 (Oct. 2003).

[6] S. Shibata, S. Honda, H. Tomiyama, and H. Takada: "Advanced system-builder: A tool set for multiprocessor design space exploration," in *Proc. 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] D. D. Gajski, N. D. Dutt, A. C-H Wu, and S. Y-L Lin: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).

[9] N. Ito, N. Ishiura, H. Tomiyama, and H. Kanbara: "High-level synthesis from programs with external interrupt handling," in *Proc. SASIMI 2015*, R1-3, pp. 10–15 (Mar. 2015).

[10] N. Ito, Y. Oosako, N. Ishiura, H. Tomiyama, and H. Kanbara: "Binary synthesis implementing external interrupt handler as independent module," in *Proc. RSP 2017*, pp. 92–98 (Oct. 2017).

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

[12] W. Nakano, Y. Shinohara, and N. Ishiura: "Full hardware implementation of FreeRTOS-based real-time systems," in *Proc. TENCON 2021*, pp. 435–440, (Dec. 2021).

[13] N. Ishiura, H. Kanbara, and H. Tomiyama: "ACAP: Binary synthesizer based on MIPS object codes," in *Proc. ITC-CSCC 2014*, pp. 725–728 (July 2014).

[14] 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. SASIMI 2022* (Oct. 2022).