Synthesis of Full Hardware Implementation of RTOS-Based Systems

Yuuki OOSAKO Nagisa ISHIURA
School of Science and Technology
Kwansei Gakuin University
Sanda, Hyogo, Japan

Hiroyuki TOMIYAMA
College of Science and Engineering
Ritsumeikan University
Kusatsu, Shiga, Japan

Hiroyuki KANBARA
ASTEM RI Kyoto
Kyoto, Japan

***Abstract*—This paper presents a method of automatically synthesizing a hardware design from a set of source codes for a real-time system utilizing an RTOS. It generates a full hardware implementation where all the tasks and handlers in the system as well as all the necessary services provided by the RTOS kernel are implemented as hardware. Every task and handler is synthesized into an independent hardware module so that it may run in parallel with the other tasks/handlers as soon as it is ready. This leads to task switching with extremely low overhead and reduced computation time both by parallel and hardware execution. Moreover, this eliminates the necessity of the task queue management; task scheduling is realized by a relatively simple manager hardware which instructs each task/handler to run or stall based on the values of its status variables. Since most of the API calls from tasks/handlers are reduced to reads/writes of these status variables, they can be expanded inline into the tasks/handlers' source codes which are compiled into hardware designs by a high-level synthesizer. We have implemented a prototype synthesis system which assume the use of the TOPPERS/ASP3 real-time kernel. A hardware implementation synthesized from a *sample1.c* code, bundled in the TOPPERS/ASP3 release, took 23 cycles for waking up a waiting task and only 1 cycle for activating an interrupt handler.**

***Index Terms*—real-time systems, RTOS, system synthesis, hardware accelerator, TOPPERS/ASP3, high-level synthesis**

I. Introduction

Today embedded systems are used in broad range of domains. Recent applications includes unmanned aerial vehicle, autonomous cars, service robots, etc. A real-time operating system (RTOS) is an indispensable component for building such real-time systems, which helps microprocessors to perform multiple tasks within the stipulated time. As the systems are deployed in more and more sophisticated and time-critical applications, it is becoming a difficult task to ensure their real-time performance.

One approach to enhancing response performance of RTOS-based systems is to implement some or all functions of RTOS in hardware. Literatures [1], [2], and [3] accelerated RTOS schedulers with hardware support. There have been even attempts to implement most of the functionalities of RTOS in hardware [4], [5], [6]. However, in this approach, tasks and handlers remain executed as software, which inevitably incurs the overhead of processor's task switching. Moreover, response time may not be improved if the heavy loads on tasks are the bottle neck.

978-1-5386-7557-1/18/$31.00

On the other hand, with high-level synthesis technology [12], the tasks and handlers are converted into dedicated hardware which leads to considerable speed-up. Automated system design methods were proposed in [7] and [8], where selected tasks or handlers were synthesized into hardware, but RTOS and some tasks remained as software executed on a CPU. Literature [9] and [10] converted a whole system including interrupt handlers into hardware, but they only dealt with bare-metal systems without an RTOS.

In this paper, we attempt to solve this problem by implementing both the RTOS functionalities and tasks/handlers into hardware. Unlike the existing RTOS acceleration methods, we get rid of scheduling queues in our method. All the tasks and handlers are implemented as independent hardware modules which may run as soon as they are ready. Then complex scheduling mechanism is reduced to a simple controller which allows/holds the execution of the tasks/handlers.

Our prototype system takes a set of C source codes implementing a real-time system using the TOPPERS/ASP3 kernel and automatically generates Verilog HDL codes. In a preliminary experiment, the generated hardware took 23 cycles for waking up a waiting task and only 1 cycle for activating an interrupt handler.

II. Real-Time Operating System

An RTOS (Real-Time Operating System) runs multiple sequential programs concurrently on a single or multiple CPUs. We refer each sequential entity a *task*. The task is initially inactive and becomes ready when certain activating conditions are met. Among the ready tasks, one of them is executed and the others are waiting for their turns. The RTOS scheduler is in charge of the choice. In real-time systems, the deadline is defined to each task and the computation of the task must be finished by the deadline. RTOSs are equipped with mechanisms to realize this. One of them is task prioritization, where a priority is set to each task and the task with the highest priority is executed. Depending on the policy of the RTOS, priority may be dynamically updated. A handler is a routine invoked to deal with events. While a cyclic handler is activated at a specified regular interval, an alarm handler is activated at a specified time. An interrupt handler is a routine invoked by

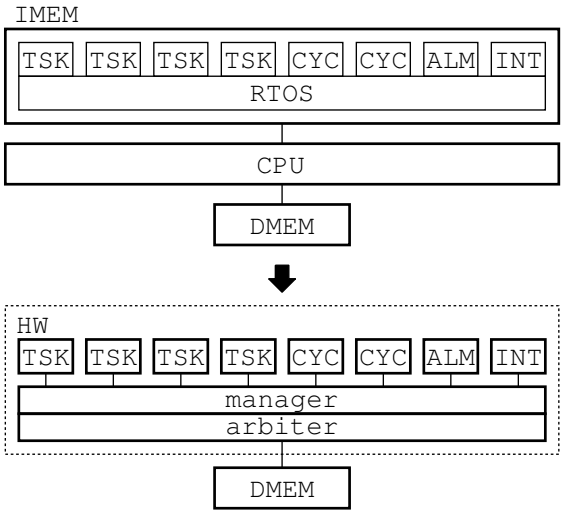

Fig. 1. Full Hardware Implementation of RTOS-Based Systems

a triggering signal or event. Usually, those handlers run in a higher priority than tasks. RTOSs provides many services other than task scheduling, such as mutual exclusion and inter-task communication.

When designing real-time systems, we must pay attention to the delays caused by the following factors:

1) Scheduling overhead, incurred by moving new and preempted tasks between the queues and selecting a job to execute.
2) Context switching overhead, incurred by saving the context (the contents of some registers) of the running task/handler, loading the context of the next task/handler, and resuming execution.
3) Waiting for CPU availability, where a ready task may have to wait for another task/handler with equal or higher priority in the running state to finish their execution.

Hardware schedulers [1], [2], [3] or hardware RTOS [4], [5], [6] can reduce the delay from the first factor, but not those from the second and the third. Some hardware support on the processor side is needed to reduce the overhead from context switching. Some parallel features are needed also on the processor side to eliminate the third overhead.

III. Full Hardware Implementation of RTOS-Based Systems

A. Overview

We propose in this paper a method of synthesizing full hardware implementation of real-time systems. Given a set of source codes of a system that runs on top of an RTOS, hardware implementation (in HDL) is auto-generated which is functionally equivalent to the system running on a CPU.

Fig. 1 illustrates the concept and a rough sketch of the resulting hardware configuration. The original system is designed to run on a CPU where binary codes of multiple tasks

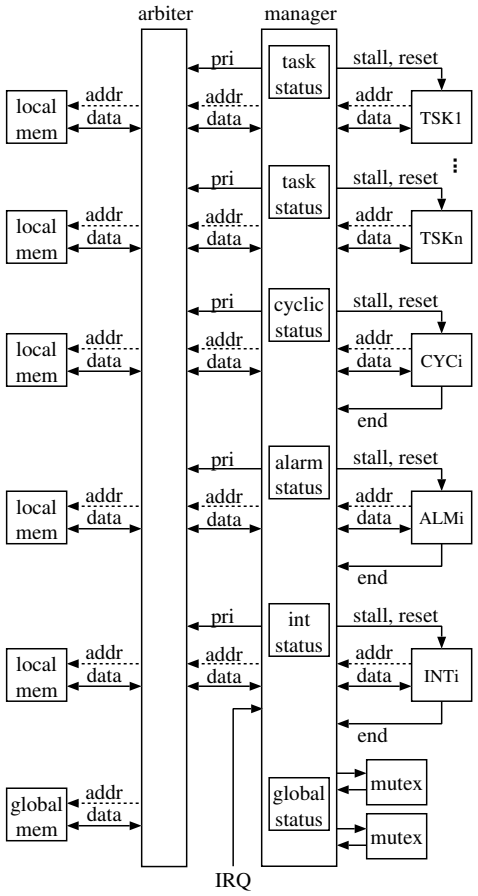

Fig. 2. Detailed configuration of synthesized hardware

(TSK), cyclic/alarm/interrupt handlers (CYC, ALM, INT), and an RTOS are stored in the instruction memory (IMEM). We assume that all the tasks/handlers are created statically at compile time rather than dynamically at run time. Then, our system converts IMEM and CPU into equivalent hardware (HW). Every task and handler is synthesized into an independent hardware module. We assume that task/handler modules which are in ready states may run in parallel regardless of their priorities, then a typical scheduler and waiting queues are eliminated. Instead, a *manager* module controls the run/stall of the task/handler modules. Since multiple accesses from the modules may come to the data memory (DMEM) at the same time, they are arbitrated by an *arbiter* module. The priorities of the tasks/handlers are reflected in this arbitration.

In this paper, we assume that mutual exclusion is well implemented in the original software version on an assumption that multiple tasks/handlers might run *in parallel*. Namely, a source code where mutual exclusion is implemented assuming that only one task/handler is running at any point of time is out of the scope of our method.

B. Hardware Configuration

Fig. 2 shows a detailed hardware configuration synthesized by our method. TSKi, CYCi, ALMi, and INTi are mod-

ules for a task, a cyclic handler, an alarm handler, and an interrupt handler, respectively. The manager module controls the tasks/handlers and the arbiter module arbitrates memory accesses.

Each task/handler has a *stall* input. When stall==1, the clock is suppressed within the task/handler. Thus the task/handler runs when stall==0, and stops when stall==1. Besides, each handler as a *reset* input which is used to restart the handler. When reset==1, the task returns to its initial state.

The manager determines the value of the stall and reset inputs based on the values of the registers to store the status of each task/handler (task status, cyclic status, alarm status, and int status in the figure). The status registers include the current status, the current priority, the waiting factor of the task/handler, for example. In the case of a cyclic and alarm handler, the remaining time until time-out is also included. When a task is in an inactive state, for example, the stall signal is set to 1, and as soon as it goes into a running state, the stall signal is cleared to 0.

The status registers are placed in the memory space. When a task/handler requests a memory access with an address, the status register is read/written if the register is mapped to the address, otherwise the request is forwarded to the memory through the arbiter. Most of the service calls are implemented as sequences of reads/writes of the status variables.

The cyclic and alarm handlers do not use external timers but have their own clock counters as part of their status registers. The counters are decremented at every clock and the handlers are activated (the stall signals are reset) when the counters become zero. As for interrupt handling, the manager watches the external interrupt signal (IRQ) and updates the status registers of the interrupt handlers, which activates the handlers.

Since all the ready tasks and handlers may run in parallel, multiple access may occur simultaneously on the same memory bank. The arbiter arbitrates the memory accesses. The arbitration is done based on the priorities of the tasks and handlers. Namely, only the memory access from the task/handler of the highest priority is granted and the stall signals are sent back to the other tasks/handlers.

Another concern regarding parallel execution of tasks/handlers is simultaneous occurrences of service calls, which may cause conflicting or inconsistent updates on status registers. To avoid this, the manager allows only one system call at a time; if there are multiple calls simultaneously, only one of them is processed first and the others are deferred. This serialization is realized using a mutex.

The mutex is also implemented in hardware. If there are multiple lock requests from tasks/handlers, only one of them is granted, and stall signals are sent back to the other tasks/handlers.

IV. Details of the method based on TOPPERS/ASP3

This section describes the details of the proposed method, based on the TOPPERS/ASP3 kernel[1] as an example.

[1]https://www.toppers.jp/en/asp-kernel.html

TABLE I
States of task in TOPPERS/ASP3

State	Meaning
Dormant	The task is not active, or its processing has ended
Ready	The task is ready and waiting for the CPU
Running	The task is being executed
Waiting	The task is suspended because some condition is not satisfied
Suspended	The task is forcibly suspended
Waiting-suspended	The task is both waiting and suspended

TABLE II
Service calls for execution control and mutual exclusion

(a) Task control

Service call	Function
act_tsk(ID tskid)	Moves the task from the dormant state to the ready state
slp_tsk()	Moves the invoking task to the waiting state
tslp_tsk(TMO timeout)	Same as slp_tsk() but with time-out
wup_tsk(ID tskid)	Cancels the waiting state of the task
rel_wai(ID tskid)	Forcibly cancels the WAITING state of the task
sus_tsk(ID tskid)	Moves the task from the running state or ready state to the suspended state, or from the waiting state to the waiting-suspended state
rsm_tsk(ID tskid)	Moves the task from the suspended state to the ready state, or from the waiting-suspended state to the waiting state
dly_tsk(RELTIM time)	Moves the invoking task to the waiting state for the specified time
ext_tsk()	Moves the invoking task from the running state to the dormant state
ras_ter(ID tskid)	Moves the task to the dormant state
ter_tsk(ID tskid)	Forcibly moves the task to the dormant state

(b) Handler control

Service call	Function
sta_cyc(ID cycid)	Moves the cyclic handler from the non-operational state to the operational state
stp_cyc(ID cycid)	Moves the cyclic handler from the operational state to the non-operational state
sta_alm(ID almid, RELTIM almtim)	Sets the activation time of the alarm handler and moves the alarm handler from the non-operational state to the operational state
stp_alm(ID almid)	Moves the alarm handler from the operational state to the non-operational state

(c) Mutual exclusion

Service call	Function
loc_mtx(ID mtxid)	Acquire the mutex
unl_mtx(ID mtxid)	Release the mutex

TOPPERS/ASP3 is the third generation real-time kernel developed in the TOPPERS project[2] based on μITRON4.0 specification [11].

The tasks are created statically at system compilation time. In the terminology of the TOPPERS kernel, a task is in one of the six states in TABLE I. The tasks are initially in the dormant state and become ready when conditions are met. The ready tasks are managed in queues and selected one is moved to the running state to be executed. The tasks may move or be forced to the waiting or the suspended (or even both) states. All the

[2]http://www.toppers.jp

TABLE III
VARIABLES TO EXPRESS TASK STATUS

Variable	Meaning
tskstat	Current status
tskpri	Current priority
tskbpri	Base priority
tskwait	Wait factor
wobjid	Waiting object ID
lefttmo	Remaining time until time-out
actcnt	Activation request count
wupcnt	Wake-up request count
raster	Termination request is raised
dister	Termination is disabled

tasks share a single memory space. The size and the starting address of the stack for each task/handlers are specified in a configuration file. TABLE II lists major service calls regarding the control of tasks and handlers. For example, in the first row, the *act_tsk* call with a task ID moves the task from the dormant state to the ready state.

A. Tasks

The stall signal to a task is set to 0 only when the task is in a running state. Otherwise (when it is either dormant, ready, waiting, suspended, waiting-suspended) the stall signal is set to 1. In addition, the stall signal is also set to 1 when requests for memory access or mutex lock from the task is deferred.

Since all the ready tasks may be executed in parallel, the tasks in the ready state immediately moves to the running state, so long as dispatch is not prohibited. This pseudo-scheduling is handled by the manager module; when some task become ready and the dispatch is not disabled, then the manager updates the state of the task from ready to running and clears the stall signal at the next clock cycle.

A running task ends its execution by calling *ext_tsk*. This moves the task into the dormant state. On notifying this, the manager sets reset = 1 and stall = 1 to bring the task to the initial state.

The status of each task is kept in task status registers in the manager module. In the ASP3 kernel the object to represent a task (in terms of the C struct) has 10 members listed in TABLE III, and we provide exactly 10 registers corresponding to the members. The status registers are mapped in the memory space and are accessed by load/store operations from the task modules. Besides of the task objects, some global status variables are also used to control the execution of the task modules. They store the interrupt prohibition flag, the interrupt mask, the dispatch prohibition flag, and the CPU lock flag.

Most of the service calls of the ASP3 kernel are implemented by a sequence of reads and updates of the task object members. This means that the original C code for the service call can be used almost as it is to generate task modules by high-level synthesis. For example, Fig. 3 is the C code to implement the act_tsk call (which moves the task from the dormant state to the ready state). The only modification from the original code is in lines 11 and 21. These calls are for acquisition and release of the mutex to serialize service calls.

```
 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. C Implementation of act_tsk

For the sake of efficiency, the body of this function is inline expanded into the tasks C code and then put into a high-level synthesizer.

B. Handlers

Cyclic and alarm handlers are dealt with basically in the same way as in the tasks; the manager controls the stall signals to the handlers based on the values of the status registers for the handlers. On the other hand, the handler notifies the completion by setting *end* signal to 1.

In the case of the cyclic handler, the manager holds two member variables per handler; *cycstat* represents the status of the handler (like tskstat of a task) where cycstat==1 means it is running and cycstat==0 means it is dormant; *lefttim* is a timer variable to keep the remaining time until the next activation (in millisecond). The cycle time is written into the timer variable when the handler is started and is decremented at each millisecond. Associated with the timer variable, the manager has a sub-counter variable that keeps the fraction of a millisecond in terms of the clock period; it is decremented at each clock cycle to measure a millisecond. When the value of the cycle timer becomes 0, the stall signal to the handler is set to 0, and the cycle time is set again to the timer variable.

The detailed flow of cyclic handler execution is as follows. The manager uses an auxiliary variable *f_reset* in addition to cycstat and lefttim, by which a request to start the cyclic handler is notified to the manager.

1) (task) calls sta_cyc(). This call sets f_reset = 1 and cycstat = 1.
2) (manager) if f_reset==1, then sets the cycle time to lefttim, and sets f_reset==0.
3) (manager) decrements lefttim at each millisecond.
4) (manager) if lefttim==0, then sets cycstat = 1 and stall = 0 to run the handler and re-sets cycle time to lefttim.
5) (handler) when the final state is reached, emits end = 1.

6) (manager) if end==1, then sets stall = 1 and reset = 1 to force the handler to the initial state (this sets cycstat = 0 and stall = 1).
7) (manager) sets reset = 0 and goto 3.

The alarm handler works in almost the same way as the cyclic handler. The difference is that the alarm handler is not repeatedly executed. Thus, the manager does not have to re-set variable lefttim.

The interrupt handler is also controlled by the same mechanism as the other types of handlers. When an interrupt is raised on the IRQ port, the stall signal is set to 0 to activate the handler, provided the interrupt is not prohibited at the point.

In the model of TOPPERS, interrupts are grouped in terms of *lines*; multiple devices of the same types are connected to a single line. When an interrupt is raised from one of the devices on a line, the interrupt service routines (ISRs) for all the devices on the line are executed sequentially. A priority is defined to each line, which may be changed dynamically, and the prohibition flag and the priority mask are provided to each line.

In our scheme, one interrupt handler module is generated per line, which includes hardware to execute the service routines of all the devices on the line.

When an interrupt is raised on a line, the manager checks the following conditions:

- The CPU lock flag is clear.
- The interrupt lock flag is clear.
- The interrupt request prohibition flag of the line is clear.
- The interrupt priority of the line is higher than the interrupt mask of the line.

If all the conditions are met, the manager sets a register *exec* representing the state of the handler to 1, which turns stall = 0 and the handler is run. The handler notifies its completion to the manager by setting end = 1. Then the manager sets reset = 1, which turns exec = 0 and stall = 1.

C. Mutex

We designed a hardware mutex module to provide the function of the mutex. If multiple locks are requested simultaneously from tasks/handlers, just one of them is granted, and stall signals are set to 1 for all the other tasks/handlers.

The detailed flow of handling the mutex is as follows. A task/handler calls loc_mtx to request for a mutex. The mutex is successfully acquired if member variable *acquired* of the mutex object is 1.

1) (task/handler i) calls loc_mtx.
2) (manager) sets the i-th request port of the mutex module LCKi to 1.
3) (mutex) sets the i-th acknowledge port ACQi to 1. If there are multiple requests on the same clock cycle, the smallest i where LCKi==1 is chosen.
4) (manager) stalls the j-th task where LCKj==1 and ACQj==0.
5) (manager) returns the value of LCKi for a read access to the member variable *acquired* from the i-th task/handler.

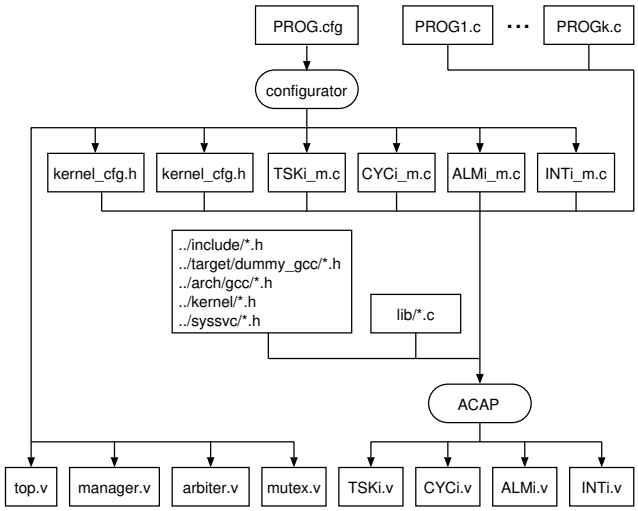

Fig. 4. Flow of system synthesis

6) (task/handler i) if acquired==1, then enters the critical section.

D. Flow of Synthesis

The tasks and handlers are compiled into register transfer level hardware by high-level synthesis, while the manager and the arbiter with the necessary number of ports are generated by a script. All the other modules such as mutex units are manually designed. The number of the tasks/handlers and necessary mutexes are defined in the configuration file, so it is extracted from the file.

V. Preliminary Result

A. Implementation

A prototype synthesis system based on the proposed method has been implemented in Perl5 utilizing a high-level synthesizer ACAP [13].

ACAP synthesizes RTL hardware design in Verilog HDL from C programs *via MIPS binaries*. Namely, given C programs are compiled and linked into a binary executable file by GCC, from which a CDFG (control dataflow graph), a popular data structure for high-level synthesis, is constructed. With this scheme, ACAP can synthesize original C codes of tasks and handlers into hardware almost with no changes. Modern high-level synthesizer such as Vivado HLS[3] may be used instead, with a little modification on the source codes.

The flow of synthesis in the implemented system is shown in Fig. 4. The names and initial settings (such as the base priorities) of the tasks/handlers are extracted from the configuration file PROG.cfg, and definition files kernel_cfg.c, kernel_cfg.h and the files containing main routines for tasks/handlers (such as TASKi_m.c)[4] are generated. Those main files are compiled with the body of the functions composed in PROG1.c, $\cdots$,

[3]https://www.xilinx.com/products/design-tools/vivado.html
[4]ACAP requires one main function per one hardware module.

TABLE IV
SYNTHESIS RESULT FOR *sample1*.

(a) Cycles and latency

service call	#cycle	latency [ns]
act_tsk	23	301.3
wup_tsk	28	366.7
ext_tsk	12	157.2
ras_ter	26	340.5
ter_tsk	23	301.3
slp_tsk	16	209.6
loc_mtx	25	327.5
unl_mtx	10	131.0
sta_cyc	19	248.9
sta_alm	20	262.0
interrupt	1	13.1

(b) Hardware size

module	#LUT	#FF
top	96	1
manager	5,305	3,027
arbiter	573	10
mutex0	29	16
mutex1	29	16
MAIN_TASK	5,382	632
EXC_TASK	4,703	399
TASK1	5,320	950
TASK2	6,620	649
TASK3	6,258	649
ALMHDR1	8,252	850
CLCHDR1	6,714	852
INTNO1	5,669	639
total	54,950	8,689

High-level synthesizer: ACAP (2016.10)
Logic synthesizer: Xilinx Vivado (2016.4)
Target: Xilinx Artix-7 (xc7a100tcsg324-3)

PROGk.c which are compiled by ACAP into hardware designs in Verilog HDL. At the same time, Verilog HDL design for the manager and the arbiter with the necessary number of ports, and the necessary number of the mutexes are generated.

At this point, we have implemented 38 out of 178 service calls of the TOPPERS/ASP3.

B. Synthesis Results

A hardware model is synthesized from sample1.c code bundled in the TOPPERS/ASP3 release. It consists of MAIN_TSK which controls the entire system, EXC_TASK which handles the CPU exception, three concurrent tasks TASK1, TASK2, TASK3, cyclic, alarm, and interrupt handlers ALMHDR1, CLCHDR1, INTNO1. The main task receives character data from a serial port that work for commands to the system, which can test the following 22 service calls:

> act_tsk, can_act, ter_tsk, chg_pri, chg_pri, chg_pri, get_pri, wup_tsk, can_wup, rel_wai, sus_tsk, rsm_tsk, ras_ter, rot_rdq, rot_rdq, rot_rdq, sta_cyc, stp_cyc, sta_alm, stp_alm, loc_cpu, and unl_cpu.

The tasks do no particular processing but just write data into the memory to indicate that they are running. The cyclic, alarm, and interrupt handlers rotates the ready queue of the highest priority to allocate CPU time to the the highest priority tasks. EXC_TASK terminates the system by calling a service call ext_ker, after outputting a log.

The generated Verilog codes are synthesized by Xilinx Vivado (2016.4), targeting a Xilinx FPGA Artix-7 (xc7a100tcsg324-3).

TABLE IV (a) shows the response performance of the synthesized hardware in terms of the clock cycles and the latency it took from the point where service call is made until the status update is completed. For example, the first row (act_tsk) indicates that it took 23 cycles to move the target task from the dormant state to the ready state (it needs one extra cycle to move the task to the running state). The update of the state alone needs just 2 cycles; the other 21 cycles were spend for error checking (validity of the task id, etc.) and for acquiring/releasing the mutex for service call serialization. The latency is the product of the cycles and the critical path delay of the circuit which was 13.098ns. All the service calls were processed less than a half micro second. Especially, activation of interrupt handler took just 1 cycle.

TABLE IV (a) (b) lists the size of each module in terms of the numbers of look-up tables (#LUT) and flip-flops (#FF). For reference, a MIPS R3000 compatible processor core takes about 3,200 LUTs. The sizes of top through mutex1 may be reasonable but those of TASK1 through INTNO1 are little too large. There seems to be a lot of room for optimizing the hardware, which we would like to work on.

VI. CONCLUSION

We have presented a method of synthesizing full hardware implementation of real-time systems using an RTOS. The synthesized hardware is able to achieve very quick response.

Currently, the size of the synthesized hardware is little too large. We are now working on reduction of the redundancy of the input source codes for high-level synthesis and optimization of the generated hardware. We are also considering the use of commercially available high-level synthesizers.

Another issue is how to apply our method to source codes that implement mutual exclusion using the fact that only one task/handler is running at any point of time. One idea to solve this problem is to provide an execution mode in which only the tasks/handlers of the highest priority are executed, or only one task/handler may run at the same time.

We are now also working on implementing other necessary service calls of TOPPERS/ASP3 regarding event flags, data queues, message buffers, etc. We are also planning to investigate the applicability of our method to other RTOSs such as FreeRTOS[5].

Acknowledgment: We would like to thank to Mr. Takayuki Nakatani who was with Ritsumeikan University, Mr. Masaharu Yano who was with Kyoto University, Mr. Shimpei Tamura who was with Kwansei Gakuin University and all the members of Ishiura Lab. of Kwansei Gakuin University for their discussion and advice on this research. This work has been partly supported by JSPS KAKENHI under Grant Nos. 16K00088, 16K01207, and 15H02680.

[5]https://www.freertos.org

REFERENCES

[1] Youngchul Cho, Sungjoo Yoo, Kiyoung Choi, Nacer-Eddine Zergainoh, and Ahmed A. Jerraya: "Scheduler implementation in MPSoC design," in Proc. Asia and South Pacific Design Automation Conference (ASP-DAC 2005), pp. 151–156 (Jan. 2005) DOI:http//doi.org/10.1109/ASPDAC.2005.1466148

[2] Melissa Vetromille, Luciano Ost, Csar A. M. Marcon, Carlos Reif, and Fabiano 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). DOI:http//doi.org/10.1109/RSP.2006.34

[3] Paul Kohout, Brinda Ganesh, and Bruce 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). DOI:http//doi.org/10.1145/944645.944656

[4] Takumi Nakano, Yoshiki Komatsudaira, Akichika Shiomi, and Masaharu 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). DOI:http//doi.org/

[5] Naotaka Maruyama, Tohru Ishihara, and Hiroto 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] Carl Stenquist: "HW-RTOS—Improved RTOS performance by implementation in silicon," White Paper—Renesas R-IN32M3 Industrial Network ASSP (May 2014). Available at available at https://www.renesas.com/en-us/media/support/partners/r-in-consortium/technology/R-IN32_HWRTOS_Whitepaper_5_20_14.pdf (accessed 2018-06-11).

[7] Seiya Shibata, Shinya Honda, Hiroyuki Tomiyama, and Hiroaki 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). DOI:http//doi.org/10.1109/SOCDC.2010.5682967

[8] Yuki Ando, Shinya Honda, Hiroaki Takada, Masato 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). DOI:http//doi.org/10.2197/ipsjjip.23.532

[9] Naoya Ito, Nagisa Ishiura, Hiroyuki Tomiyama, and Hiroyuki Kanbara: "High-level synthesis from programs with external interrupt handling," in *Proc. Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2015)*, 10–15 (March 2015).

[10] Naoya Ito, Yuuki Oosako, Nagisa Ishiura, and Hiroyuki Tomiyama, and Hiroyuki Kanbara: "Binary synthesis implementing external interrupt handler as independent module," in Proc. International Symposium on Rapid System Prototyping (RSP 2017), pp. 92–98 (Oct. 2017). DOI:http//doi.org/10.1145/3130265.3130317

[11] ITRON Committee, TRON association: *μITRON4.0 Specification* (1999, 2002). Available at http://www.ertl.jp/ITRON/SPEC/FILE/mitron-400e.pdf (accessed 2018-06-11).

[12] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu, and Steve Y-L Lin: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).

[13] Nagisa Ishiura, Hiroyuki Kanbara, and Hiroyuki Tomiyama: "ACAP: Binary Synthesizer Based on MIPS Object Codes," in *Proc. ITC-CSCC 2014*, pp. 725–728 (July 2014).