Parallel Execution of RTOS Services in Full Hardware Implementation of RTOS-Based Systems

Yuki Nakatani
Graduate School of Science and Technology
Kwansei Gakuin University
Sanda, Hyogo, Japan
nakatani-yuki@kwansei.ac.jp

Nagisa Ishiura
School of Engineering
Kwansei Gakuin University
Sanda, Hyogo, Japan
ishiura@kwansei.ac.jp

***Abstract*—This paper proposes a method for parallel execution of RTOS services in a full hardware implementation that synthesizes an entire real-time system, including RTOS functionality, into hardware. In conventional full hardware implementations of RTOS-based systems, tasks are executed in parallel by independent hardware modules; however, RTOS services are processed sequentially to prevent interference among services. We observe that certain combinations of RTOS services do not interfere with each other and therefore can be executed concurrently. When multiple service requests are issued by tasks, the proposed method examines whether any requests do not conflict with the services currently being executed. If such requests exist, the method selects and dispatches one from the highest-priority task among them. Interference is determined based on two conditions: whether a control service that may update task states is currently executing, and whether the service module responsible for the requested service is already occupied. The proposed hardware was designed in Verilog HDL. Experimental results demonstrate that, although the circuit size increases, the proposed method significantly reduces the number of execution cycles when multiple tasks issue service requests simultaneously.**

***Index Terms*—hardware implementation of real-time systems, RTOS, real-time operating system, TOPPERS**

I. Introduction

In recent years, embedded systems deployed in applications such as automotive equipment and unmanned aerial vehicles have faced rapidly increasing computational demands as their functionalities become more sophisticated. Such systems, which must satisfy strict real-time requirements, are commonly designed using a real-time operating system (RTOS); however, ensuring adequate responsiveness is becoming increasingly difficult due to the growing complexity of control and the increasing workload.

Many studies have proposed implementing RTOS functions in hardware to improve the responsiveness of RTOS-based systems. In [1], the RTOS scheduler is implemented in hardware, and [2] implements the entire RTOS in hardware. However, because tasks and handlers are still executed in software in these approaches, overheads due to CPU waiting and context switches remain.

In contrast, [3] proposes a method in which not only the RTOS functions but also all tasks are implemented entirely in hardware. In this approach, each task is implemented as an independent hardware module, and all tasks that become ready can execute in parallel; therefore, no overhead is incurred from CPU waiting or task switching. Furthermore, [4] proposes a method that generates hardware directly from task source code using a high-level synthesis tool.

In [4], the hardware that realizes RTOS functionality performs not only scheduling but also updates task states and executes RTOS services such as mutexes and data queues in hardware [5]. While tasks execute in parallel, these RTOS services are executed sequentially to avoid interference among services, which leads to increased response time when many tasks issue service requests.

To address this issue, this paper proposes a method for executing eligible RTOS services in parallel. For each service request issued by a task, parallel execution is permitted if the request is determined not to interfere with service processing already in execution. To suppress hardware growth, we do not replicate the modules responsible for accepting and executing services.

We implemented the proposed method in Verilog HDL. Although the circuit size increases, experimental results confirm that the proposed method can reduce the overall execution time in scenarios where service requests are issued from multiple tasks.

II. Full Hardware Implementation of RTOS-Based Systems

A. Full Hardware Implementation of RTOS-Based Systems

Oosako *et al.* proposed a method that implements both RTOS functions and all tasks/handlers entirely in hardware [3]. Figure 1 illustrates the concept. The upper part shows a typical software implementation of a real-time system, where the RTOS and the tasks (TSK_i) running on top of it are executed by a CPU. The lower part shows the full hardware implementation of the same system. Each task (TSK_i) is synthesized into an independent hardware module by high-level synthesis. The *manager* is a hardware module that provides RTOS functionality, including task execution control and RTOS services such as inter-task synchronization and communication (e.g., mutexes and data queues).

In this system, all tasks that have transitioned to the ready state execute in parallel. As a result, overheads due to CPU

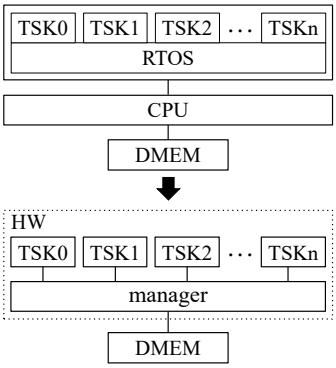


Fig. 1: Full hardware implementation of RTOS-based system [3]

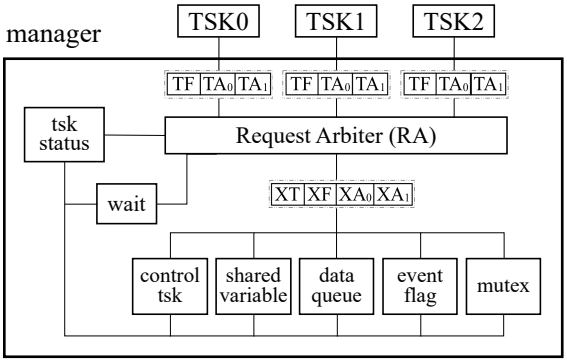


Fig. 2: Architecture in [4]

waiting, scheduling, and context switching are eliminated. Moreover, because tasks can run faster when implemented in hardware, the system responsiveness can be significantly improved.

B. Architecture Proposed by Ando and Muguruma

Ando and Muguruma refined the architecture in [3] to substantially reduce both hardware area and execution time, and to enable hardware implementation of tasks using commercially available, general-purpose high-level synthesis (HLS) tools [4].

Figure 2 shows the hardware organization. TSK0–TSK2 are task modules synthesized by HLS, and *manager* is a hardware module implementing RTOS functionality.

tsk_status is a set of state registers that holds the state and priority of each task as well as the global kernel state. Based on this information, the manager controls whether each task is allowed to run or is halted.

The modules placed in the lower part of the manager—*control_tsk*, *shared_variable*, *dataqueue*, *event_flag*, and *mutex*—are *service modules* that execute RTOS services. The *control_tsk* module handles service calls related to task management, such as updating task states and priorities. The *shared_variable* module provides access to variables shared among tasks. The *dataqueue*, *event_flag*, and *mutex* modules provide inter-task data transfer and mutual exclusion.

A task's service call is translated into writes from the task module to the control registers TF, TA0, and TA1. When a task writes the requested service ID to TF and writes arguments to TA0 and TA1, the manager invokes the corresponding service module to execute the requested operation.

In this architecture, all ready tasks execute in parallel, whereas service-call processing is performed sequentially to avoid interference among services. When multiple tasks request services at the same time, the Request Arbiter (RA) selects the request issued by the highest-priority task and dispatches it for execution.

The service-call execution flow is as follows:

1) A task requests a service by writing to TF, TA0, and TA1, and then waits for completion.
2) Among the tasks requesting services, the RA selects the highest-priority one, writes its task ID to XT, and copies the task's TF, TA0, and TA1 into XF, XA0, and XA1, respectively.
3) The service module responsible for the request indicated by XF performs the operation based on XF, XA0, and XA1, and returns a result (error code) by writing it to XA0.
4) The manager writes the value in XA0 back to TA0 of the requesting task.
5) After receiving TA0, the task resumes execution.

Although each service module completes its operation in one or two cycles, it still takes at least five cycles from when a task issues a service request until the request completes. Although service requests (Step 1) can be accepted in parallel, all subsequent processing is serialized; thus, the next service cannot begin until the current one finishes. Consequently, when service calls are concentrated, the waiting time can increase significantly, potentially degrading system responsiveness.

III. Parallel Execution of RTOS Services

A. Overview

This paper proposes a method for executing RTOS services in parallel in order to further improve the responsiveness of the hardware-implemented RTOS-based system proposed in [4].

When multiple service requests are issued by different tasks, our method executes in parallel those requests that (i) are handled by different service modules and (ii) do not interfere with each other. While service request acceptance and service execution proceed concurrently, dispatching requests to service modules is performed sequentially so as to prevent excessive growth of the control hardware.

B. Services Eligible for Parallel Execution

In this paper, "interference among RTOS services" refers to a situation in which the concurrent execution of multiple RTOS services leads to conflicting updates to the state or priority of the same task, potentially producing results that differ from those obtained under sequential execution. For example, in TOPPERS/ASP, the service call `ras_ter(ID`

t) issues a termination request to task t, and the manager updates the state of task t to the dormant state according to that request. On the other hand, when task t executes a mutex-lock service (e.g., `loc_mtx`), if the mutex cannot be acquired, the state of task t is updated to a waiting state. If these services progress in parallel, their updates to the state of the same task t may conflict. To avoid such conflicts, [4] serializes service processing when multiple service requests exist.

However, for services that do not update the state of tasks other than the calling task, parallel execution yields the same result as sequential execution. For instance, while task A is executing a dataqueue send service (e.g., `snd_dtq`), another task B may execute a mutex-lock service. Because these services do not update the states of tasks other than the currently running tasks, no conflict arises in task-state updates even when the services are executed in parallel.

Therefore, executing non-interfering RTOS services concurrently can improve overall throughput and system responsiveness.

At the same time, suppressing hardware growth is also crucial. For example, two tasks requesting locks on two different mutex objects do not interfere and could be executed in parallel; however, realizing this would require duplicating the mutex service module, which increases hardware area. Moreover, enabling parallel execution for all combinations of services (and their arguments) would incur a high cost for determining whether a given combination is safe.

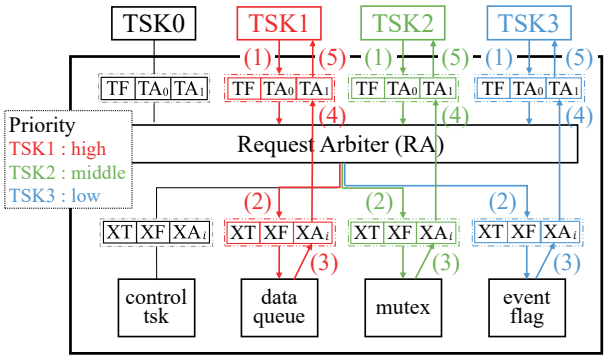
Accordingly, to prevent interference while keeping the hardware size small, we determine parallel executability based on the following criteria. We do not replicate service modules; thus, at most one service handled by the same service module can be executed at a time. In addition, services executed by *control_tsk* cannot be executed in parallel with any other service, because they may update the state of tasks other than the calling task.

C. Architecture for Parallel Service Execution

Because the hardware for arbitrating service requests is large and grows in proportion to the number of tasks, replicating the arbitration logic would significantly bloat the manager. Therefore, our method adopts the following approach: a *single* arbiter selects requests sequentially, while non-interfering services are allowed to execute in parallel.

In the conventional method, when multiple service requests are issued from tasks, the arbiter selects one request and dispatches it to the corresponding service module; only after the service processing completes and the requesting task receives the result does the next service begin. In contrast, our method starts executing a non-interfering service *without waiting* for the previously dispatched service to finish.

In the conventional arbitration, the request issued by the highest-priority task is always selected. In our method, among the requests that do not interfere with services currently in execution, the arbiter selects the one issued by the highest-priority task.



(a) Hardware configuration for the proposed method

Clock	1	2	3	4	5	6	7	8	9	10	11	12	13	14
TSK1 (loc_mtx)	(1)	(2)	(2)	(3)	(4)	(5)								
TSK2 (snd_dtq)	(1)	(2)					(2)	(3)	(4)	(5)				
TSK3 (loc_mtx)	(1)	(2)									(2)	(3)	(4)	(5)

(b) Service execution flow in the conventional method

Clock	1	2	3	4	5	6	7	8	9	10
TSK1 (loc_mtx)	(1)	(2)	(2)	(3)	(4)	(5)				
TSK2 (snd_dtq)	(1)	(2)			(2)	(3)	(4)	(5)		
TSK3 (loc_mtx)	(1)	(2)					(2)	(3)	(4)	(5)

(c) Service execution flow in the proposed method

Fig. 3: Parallel execution of RTOS services

Figure 3(a) shows the manager configuration in the proposed method. The registers XT, XF, and XAi are replicated for each service module, enabling the Request Arbiter to activate multiple service modules in parallel. The Request Arbiter itself is not duplicated, although the service selection logic is modified.

Suppose that TSK1 requests a dataqueue service (`snd_dtq`), TSK2 requests a mutex service (`loc_mtx`), and TSK3 requests an event-flag service (`set_flg`) simultaneously. Their priorities are assumed to be in descending order of TSK1, TSK2, and then TSK3.

Figure 3(b) illustrates the service execution flow in the conventional method. Service requests are selected one by one by the arbiter, and the next service is not started until the current service has finished processing and returning its result. Consequently, the service requested by the lowest-priority task, TSK3, does not begin until the services requested by TSK1 and TSK2 complete. It takes 14 cycles to complete the three services.

Figure 3(c) illustrates the execution flow in the proposed method. A service request that does not interfere with services currently in execution is dispatched and executed without waiting for those running services to complete. The execution cycles are reduced to 10.

TABLE I: Execution cycles for processing service requests

	Conventional	Proposed
T1 (act_tsk), T2 (snd_dtq), T3 (loc_mtx)	15	12
T1 (loc_mtx), T2 (chg_pri), T3 (snd_dtq)	15	13
T1 (loc_mtx), T2 (snd_dtq), T3 (loc_mtx)	14	10

D. Selecting Services to Execute

The determination of executable services in the arbitration circuit, when multiple service requests are issued, is performed based on information indicating which service modules are currently active. In each service module, the service ID is stored in the intermediate register XF while the corresponding service is being processed. Therefore, by monitoring XF, the execution status of each service module can be identified. Based on this information, only service requests that do not interfere with the services currently in execution are forwarded to the arbitration circuit.

A service request r is masked at the arbiter input stage and excluded from arbitration if any of the following conditions holds:

- The *control_tsk* module is currently executing.
- Request r would be handled by *control_tsk*, and at least one other service module is currently executing.
- The service module that would process request r is already executing a service.

Among the service requests not masked by the above conditions, the arbiter selects and dispatches the one whose requesting task has the highest priority.

IV. Experimental Results

We designed the *manager* module in Verilog HDL based on the proposed method.

We evaluated the number of cycles required when three tasks (T1, T2, and T3) simultaneously issue service calls in a configuration with three service modules (*control_tsk*, *mutex*, and *dataqueue*).

Table I summarizes the results. In the first row, `act_tsk`, `snd_dtq`, and `loc_mtx` denote service calls for task activation, sending data to a data queue, and requesting a mutex lock, respectively. The conventional method required 15 cycles, whereas the proposed method reduced this to 12 cycles. In the second row, `chg_pri` changes a task priority. In the third row, two tasks request mutex locks simultaneously. In all cases, the proposed method reduces the execution cycles.

We synthesized the designed *manager* using Xilinx Vivado 2023.2, targeting a Xilinx Artix-7 FPGA (xc7a100tcsg324-3). This *manager* includes three service modules (*control_tsk*, *mutex*, and *dataqueue*) and controls eight tasks as well as two instances each of alarm handlers, cyclic handlers, and interrupt handlers.

Table II shows the synthesis results. #LUT and #FF denote the numbers of LUTs and flip-flops, respectively, and *Delay* is the critical-path delay. Although the circuit size increases by approximately 2.75×, the delay remains below 10 ns.

TABLE II: Logic synthesis results of the manager module

	#LUT	#FF	Delay [ns]
Conventional [4]	5,023	2,002	6.55
Proposed	13,827	2,293	8.49

V. Conclusion

This paper proposed a method for executing RTOS services in parallel in the full hardware implementation of RTOS-based systems. While keeping service-request arbitration sequential, the proposed method enables parallel service processing by dispatching only those service requests that do not interfere with services currently in execution. We experimentally observed that the proposed method can reduce the execution cycles in scenarios where multiple tasks issue service requests simultaneously.

In the current implementation, however, the circuit size and critical-path delay are larger than we originally expected. We are currently investigating the causes of these overheads and working to reduce both the circuit size and the delay to reasonable levels. Furthermore, the current policy for avoiding interference among service executions is somewhat conservative. Therefore, another important future direction is to explore methods for achieving more parallel service execution while suppressing increases in circuit size and delay. Evaluating the proposed method with more practical-scale systems and applications is also left for future work.

Acknowledgements

Authors would like to express their appreciation to Dr. Hiroyuki Kanbara and Prof. Hiroyuki Tomiyama of Ritsumeikan University, and Mr. Takayuki Nakatani (formerly with Ritsumeikan University) for their valuable advises. We would also like to thank to the members of Ishiura Lab. of Kwansei Gakuin University for their discussion.

References

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

[2] 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).

[3] 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).

[4] T. Ando, I. Muguruma, Y. Ishii, N. Ishiura, H. Tomiyama, and H. Kambara: "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).

[5] 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).