Real-Time Operating Systems for Multicore Embedded Systems

Hiroyuki Tomiyama Shinya Honda Hiroaki Takada
Graduate School of Information Science
Nagoya University
Furo-cho, Chikusa-ku, Nagoya 464-8603, Japan

Abstract — **Multicore systems-on-chip have become popular in the design of embedded systems in order to simultaneously achieve high performance and low power consumption. On the software side, real-time operating systems are necessary in order to handle growing complexity of embedded software. This paper describes requirements, design principles and implementation techniques for real-time operating systems to be used in asymmetric multicore systems.**

Keywords – real-time operating systems; asymmetric multprocessors; embedded systems

I. INTRODUCTION

Multiprocessor systems-on-chip (MPSoCs) are now employed in a variety of embedded systems such as cellular phones and multimedia devices. This is mainly because they simultaneously offer high performance, low cost and low power consumption.

MPSoCs are broadly classified into three types. The first type is symmetric shared-memory MPSoCs, where the processors are homogeneous, and all the resources including main memory and peripherals are shared by the processors. In many cases, symmetric MPSoCs have coherent caches, and an operating system (OS) dynamically allocates tasks (or threads) onto the processors at runtime in order to balance the loads over the processors. Therefore, software programming for symmetric MPSoCs are relatively easy. Symmetric multiprocessors are preferable in high-performance computers where average performance (or throughput) is more important than guaranteed response times. However, bounding worst-case performance is very difficult since more shared resources generally lead to higher possibility of resource conflicts. Therefore, symmetric multiprocessors are rarely used in hard real-time systems. The second type is asymmetric MPSoCs. Processors may be homogeneous or heterogeneous. Each processor has local memory, which can be accessed by the other processors at the cost of longer access time. Recently, asymmetric heterogeneous MPSoCs are widely used in embedded systems where a set of application tasks are fixed at a design time. The tasks are statically allocated onto the processors, and the processors and their peripherals are optimized for the allocated tasks. The static task allocation policy together with dedicated peripherals and limited resource sharing makes it easier to bound worst-case performance compared with symmetric MPSoCs, and also improve, and power and performance scalability can be improved. On the negative side, software programming is more difficult. The last type is message-passing MPSoCs, where the processors do not share main memory. Each processor has private memory which cannot be accessed by the other processors. In order for tasks running on different processors to communicate with each other, message packets are transmitted over the interconnection network. The message-passing MPSoCs are often called network-on-chips (NoCs). On the software side, tasks are statically allocated onto the processors. An OS runs on each processor, and inter-processor communication is managed at the middleware or application level. Message-passing MPSoCs represent good performance scalability for applications with few communications, but not for applications which frequently interact with each other due to the large communication overhead.

There exist a number of commercial and research-purpose RTOSs for symmetric MPSoCs (hereinafter, referred to as SMP-RTOSs). For message-passing MPSoCs, OSs for uniprocessor systems can be used. On the other hand, RTOS technology for asymmetric MPSoCs has not been established well. Traditionally, RTOSs for uniprocessor systems were used for asymmetric MPSoCs. In this case, inter-processor communications are realized at a middleware level or an application level. In the former case, communication overhead is not trivial. In the latter case, programming application tasks are not easy, and rewriting is necessary every time we want to explore different task allocations at the design time. Another traditional way is to use SMP-RTOSs for asymmetric MPSoCs. Many of SMP-RTOSs offer the functionality to allocate specific tasks onto specific processors. Using this functionality, the SMP-RTOSs can be used for asymmetric MPSoCs. However, due to the internal data structure of SMP-RTOSs, inter-processor conflicts may often happen within the SMP-RTOSs [1], which results in degradation of the worst-case performance. Thus, neither uniprocessor RTOSs nor SMP-RTOSs are appropriate for asymmetric MPSoCs to be used in real-time embedded systems.

We have developed an RTOS, named TOPPERS/FDMP Kernel, for asymmetric MPSoCs. The FDMP kernel is based

[1] For example, if tasks are maintained with a single ready queue within the SMP-RTOS, the ready queue should be accessed exclusively. Then, when multiple processors need to access the ready queue at the same time, a conflict occurs.

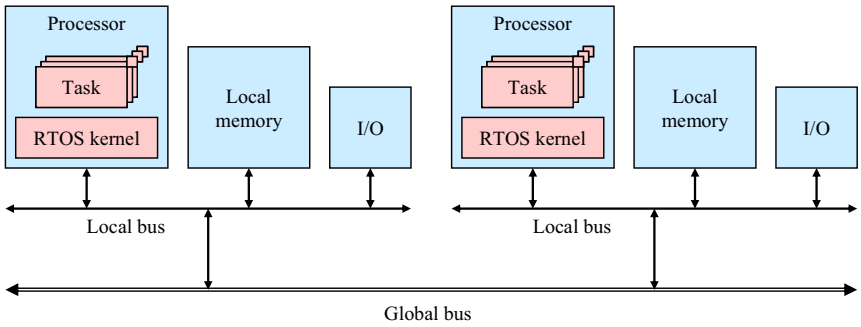

Figure 1. Target system architecture

on the µITRON specification[2]. µITRON is one of the most popular RTOSs in many Asia and Pacific countries, specifically in Japan, because of its quick response time and small code size. However, µITRON does not explicitly support multiprocessor systems, so the FDMP kernel slightly extends the µITRON specification towards asymmetric MPSoCs. Based on the FDMP kernel, we have also developed another RTOS, named TOPPERS/FMP Kernel, which explicitly supports dynamic task migration across processors.

This paper describes TOPPERS/FDMP Kernel and TOPPERS/FMP Kernel. In Section II, we discuss requirements and principles in the design of RTOSs for asymmetric MPSoCs. Section III presents several techniques to implement the FDMP kernel. Section IV shows performance evaluation of the FDMP kernel. Then, in Section V, the FMP kernel is outlined.

II. Requirements and Principles

This section discusses requirements and principles for extending the µITRON specification towards asymmetric MPSoCs.

A. Target System Architecture

In this paper, we assume asymmetric shared-memory MPSoCs to be used in small- to mid-scale real-time systems. In such systems, memory capacity is small, and virtual memory mechanism is not employed in order to achieve quick responses. Figure 1 shows the target system architecture. Each processor has local memory and external I/O interface such as UART, which are connected by a local bus. The local memory can be accessed by the other processors through a global bus. A processor can send interrupts to any of the other processors. There exists a hardware support (such as hardware semaphores and test&set instruction) to realize mutual exclusion among the processors.

Tasks and interrupt handlers are statically allocated to processors at the design time in order to minimize inter-processor communication and maximize the independence of the individual processors.

B. Requirements for RTOSs

RTOSs for asymmetric MPSoCs should satisfy the following requirements.

1. System calls for inter-processor communication should be compatible to ones for intra-processor communication. This is a very important requirement when we want to reuse application tasks which were written for necessary systems. Also, this requirement is desirable when we want to explore different task allocation alternatives. Unless this requirement is met, the application tasks need to be modified every time tasks are reallocated.
2. Tasks and interrupt handlers must not be interfered by those running on different processors as long as they do not execute inter-processor communication system calls. Also, worst-case execution time of the tasks and worst-case response time of the interrupt handlers must be bounded independent of the number of processors. This requirement assures the scalability against the number of processors. Most SMP-RTOSs do not satisfy this requirement since inter-processor conflicts may occur within the RTOSs even if no inter-processor communication system call is executed.
3. Worst-case response time for inter-task communication system calls must be bounded although the worst-case response time inevitably depends on, i.e., at least proportional to, the number of processors.

C. Overview o f µITRON 4.0 and its Extension for Asymmetric MPSoCs

Before we discuss how to extend the µITRON specification towards asymmetric MPSoCs, we briefly describe the µITRON specification which was designed for uniprocessor systems.

µITRON is a standardized specification of RTOS APIs for small- to mid-scale real-time systems with typical code size of about 20K bytes. Typical application domains include consumer electronic products, mobile devices, and automotive electronic control systems. µITRON has been developed and

[2] To be more precise, µITRON is not a name of specific RTOS product, but is a name of standardized specification of RTOS. Therefore, there exist a number of µITRON-compliant RTOSs in the world.

refined over more than two decades, and is the most popular RTOS specification in Japan with the market share of approximately 50%. The latest release at present is µITRON 4.0. µITRON defines several profiles, and the most fundamental one is called *Standard Profile*.

µITRON employs a priority-based preemptive scheduling policy, and the priority of tasks can be changed at runtime. µITRON 4.0 Standard Profile states that the number of priority levels must be at least 16. µITRON supports several basic synchronization and communication mechanisms, such as semaphores, events, data queues, and mailboxes. Dynamic memory allocation using so-called memory pools is supported in µITRON. Since µITRON is designed for small embedded systems, virtual memory, dynamic module loading or memory protection mechanism is not supported. All the tasks are linked together with the RTOS kernel code to generate a single object module. Kernel objects such as tasks and semaphores are statically instantiated by means of *static APIs* in a *confugulation file*. The configuration file is fed by so-called *configurator* to generate C files where the objects are instantiated and initialized. The C files are compiled and linked with application tasks as well as the RTOS kernel code.

Interrupt handling is one of important mechanisms in RTOSs. µITRON provides a number of API functions for defining interrupt handlers, allowing and prohibiting interrupts, changing interrupt masks, and so on. The number of interrupt levels is not determined by µITRON but implementation-dependent. µITRON has three types of time event handlers, i.e., cyclic handlers, alarm handlers, and overrun handlers. Cyclic handlers are invoked periodically, alarm handlers are invoked at a specified time, and overrun handlers are invoked when the execution time of a task exceeds a specified time.

In addition, µITRON provides a number of services which are necessary for industrial use. More detailed documents on µITRON are found in [1].

We have extended the µITRON specification towards asymmetric MPSoCs. Specifically, the followings are modified.

- Classification of objects
- Identification of objects
- System states
- Static APIs

D. Classification of Objects

Each kernel object (such as task and semaphore) belongs to one of the processors. A set of kernel objects which belong to the same processor is called a *class*. Allocation of the kernel objects to processors is statically defined in the configuration file. Identification (ID) numbers are given to the classes.

The following objects can be executed only on the processor to which the objects belong.

- Application tasks
- Task exception handling routines
- Cyclic handlers

```
local class CPU1 {
        CRE TSK(TASK1, {TA HLNG, …});
        CRE TSK(TASK2, {TA HLNG, …});
        CRE CYC(CYCHDR1, {TA HLNG, …});
}
local class CPU2 {
        CRE TSK(TASK3, {TA HLNG, …});
        CRE TSK(TASK4, {TA HLNG, …});
        CRE CYC(CYCHDR2, {TA HLNG, …});
}
```

Figure 2. Fragments of a configuration file for a dual-processor system

- Interrupt handlers
- CPU exception handlers

Due to this, the RTOS can perform task scheduling independently of the other processors.

E. Identification of Objects

According to the µITRON specification, unique ID numbers are given to kernel objects, and system calls which manipulate an object have its ID number as an argument. In the extended µITRON, an object has a unique ID number of 32 bits, and the ID number consists of two parts. The upper 16 bits are used to specify a class ID number, and the lower 16 bits are used to specify an ID number in the class. A class ID of zero means that the object belongs to the same processor on which the system call is issued. In this way, µITRON system calls can be used without changing their APIs.

F. System States

µITRON 4.0 defines system states for exclusively executing a specific task. System states include *locked CPU state* where no interrupt or task switch is permitted and *suppressed dispatch state* where no task switch is permitted. These states are often used for mutual exclusion.

In the extended µITRON, the system state is controlled processor-by-processor independently. For example, if one of the processors is in the locked CPU state, interrupts and task switches can be allowed on other processors. Therefore, mutual exclusion across processors cannot be realized by using these system states. With the extended µITRON, mutual exclusion should be implemented explicitly using synchronization objects (such as semaphores). If we want to reuse software with the state-based mutual exclusion mechanism, the software needs to be rewritten.

The reason why the system state is defined for each processor instead of for the entire system is to satisfy the second requirement shown in Section II.B.

G. Static APIs

As mentioned in Section II C, kernel objects are statically instantiated and dynamic instantiation of kernel objects at runtime is not supported. Kernel objects are defined by means of static API in a configuration file, which is then fed by a configurator to generate C files.

In the extended μITRON, syntax of the configuration file has been extended in order to specify allocation of kernel objects to specific processors. Figure 2 shows a fragment of a configuration file for a system with two processors CPU1 and CPU2. For each processor, two tasks and one cyclic hander are instantiated. As shown in Figure 2, changing allocation of tasks and other objects is very easy. For example, if we want to reallocate TASK3 from CPU2 to CPU1, the only thing to do is to move the corresponding line from CPU2 to CPU1 in the configuration file.

III. IMPLEMENTATION TECHNIQUES

This section describes several techniques to develop TOPPERS/FDMP Kernel which implements the extended μITRON specification presented in the previous section.

A. Independent Control Blocks

Data structures for controlling kernel objects are called *control blocks*. In the FDMP kernel, kernel objects are statically allocated to specific processors, and therefore, control blocks are also statically allocated to the processors. In other words, a processor has its own control block for the kernel objects allocated to the processor. The kernel objects and the control block are placed on the local memory of the processor. Thus, as long as inter-processor communication is not called, tasks are not interfered by other processors (i.e., the second requirement in Section II.B).

B. Inter-Processor System Calls

There exist broadly two methods to realization of inter-processor system calls. One method is *direct manipulation*, where the processor directly accesses the control block of the remote processor[3]. The other method is based on *remote call*, where a processor sends a request to the remote processor. The remote call method is applicable not only to asymmetric multiprocessors but also to message-passing multiprocessors without shared memory.

The FDMP kernel employs the direct manipulation method due to its low overhead of performance.

C. Lock Units

The FDMP kernel realizes inter-processor system calls by directly manipulating the control block of the remote processor. This manipulation requires mutual exclusion among the processors. The FDMP kernel employs spinlocks for the mutual exclusion.

A *lock unit* denotes a set of resources which are controlled by a lock for mutual exclusion. The size of lock units significantly affects the scalability and the response time of system calls. Larger lock units lead to more inter-processor conflicts. For example, if all the resources in the system are controlled by a single lock, called a *giant lock*, conflicts occur even when no inter-processor system call is executed. On the other hand, if the lock unit is too fine-grained, multiple locks need to be acquired in order to execute a system call, which results in degradation of response time or unexpected deadlock.

We have carefully analyzed the internal structure of system calls, and then we have decided the lock units as follows. For each processor, two locks are defined. One is called a *task lock*, while the other is an *object lock*. The task lock is used for mutual exclusion of data structures related to task control. Such data structures include task control blocks (TCBs). On the other hand, the object lock is used for mutual exclusion of data structures related to communication and synchronization. Such data structures include semaphores, event flags and data queues. In order to avoid deadlock within the RTOS, we have defined the order of lock acquisition as the object lock first and then the task lock.

D. DeadLock Avoidance

If all of system calls follow the order of lock acquisition, no deadlock occurs. However, some system calls cannot follow this by their nature. For example, it is not determined which object control block should be accessed before accessing the task control block. In such system calls, a task lock needs to be acquired first and then an object lock.

In the FDMP kernel, such system calls are implemented as follows. First, the task lock is acquired, and the object control block to be accessed is identified. At this point, the task lock is released. Then, the object lock is acquired, and finally the task lock is acquired.

However, at the time the task lock is released, other task may modify the task control block. One way to avoid such inconsistency is to release both object lock and task lock, and then acquire the object lock and task lock in this order. This simple retrial-based solution does not bound the worst-case response time and does not satisfy the second and third requirements in Section II.B.

In order to avoid unbounded retrial, the FDMP kernel is implemented as follows. Before releasing the task lock, a flag is set which denote that the task control block needs to be modified soon. At this point, a different new task acquires the task lock. The new task finds the flag, and then, on behalf of the previous task, the new task performs the operation which needs to be done by the previous task, and then clears the flag. When the previous task gets the task lock again, it finds the flag cleared and know that the operation was already done by other task. In this way, the FDMP kernel avoids the deadlock without unbounded retrial.

E. Inter- and Intra-Processor Mutual Exclusion

For execution of some system calls, both inter-processor mutual exclusion and intra-processor one are necessary. The FDMP kernel realizes inter-processor mutual exclusion using spinlock and realizes intra-processor one by disabling interrupts. These two are related with each other, and special care is needed in order to guarantee the worst-case response.

For example, if we acquire the inter-processor lock first and then disable interrupts, we may accept an interrupt request which arrives between acquiring the inter-processor lock and disabling interrupts. During the interrupt handling, other

[3] As mentioned in Section II.A, a processor can access the memory of other processors.

```
retry:

// acquire object lock
disable_interrupt();
while (test_and_set(obj_lock) == LOCKED) {
        if (interrupt_request() == TRUE) {
                enable_interrupt();
                goto retry;
        }
}

// Since the code here may be executed more than once,
// it must not modify the data inside the kernel.

// acquire task lock
while (test_and_set(tsk_lock) == LOCKED) {
        if (interrupt_request() == TRUE) {
                release_lock(obj_lock);
                enable_interrupt();
                goto retry;
        }
}

// Critical section
```

Figure 3. Intra- and inter-process mutual exclusion using test&set lock

processors remain waiting, which is a waste of time. On the contrary, if we disable interrupts first and then acquire the inter-processor lock, interrupts remain disabled for a long time, which degrade the interrupt response time.

It is ideal that the processors offer a hardware mechanism which performs both inter-processor and intra-processor mutual exclusion in an atomic manner, but unfortunately, no such processor exist at present.

Based on the work in [2], the FDMP kernel solves this problem as shown in Figure 3. First, interrupts are disabled for intra-processor mutual exclusion, and then the test&set lock is acquired for inter-processor mutual exclusion. While waiting for the test&lock to be acquired, we check if an interrupt request arrives. If arrived, it is accepted.

When the object lock is already acquired and the task lock needs to be acquired, the FDMP kernel works as follows. If an interrupt request arrives while waiting for the task lock, the object lock which was already acquired is released, and then the interrupt is accepted. After the interrupt handling, the FDMP kernel retries from the beginning, i.e., acquiring the object lock.

In this way, the FDMP kernel realizes inter- and intra-processor mutual exclusion. However, with the test&set method, the worst-case waiting time is not bounded inevitably. Bounding the response time is one of our future work.

IV. EVALUATION

We have conducted a set of experiments to evaluate the usefulness of TOPPERS/FDMP Kernel. Altera NiosII/s, which is a soft-core processor for FPGA, is used as a target processor. We have developed a multiprocessor platform with four NiosII/s processors. Each processor has a local memory. The Avalon bus, which is a standard bus for NiosII systems, is based on a star-type network, so no contention happens as long as the processors access their local memories. All the components (i.e., processors, memories, bus, etc.) operate at 50MHz.

A. Code Size

Code size of the FDMP kernel is compared with TOPPERS/JSP Kernel which is a µITRON-compliant RTOS for uniprocessor systems developed by us. The result is summarized in Table 1. The size of the text section for the FDMP kernel is about 60% larger than that for the JSP kernel. One of the reasons for the increased code size is that, for each system call, a routine for acquiring and releasing a lock is inserted. Also, a new routine for avoiding deadlocks is added. On the other hand, an increase in the data and bss secsions is trivial. An increase in data size is also small. A new data block, named CCB (Class Control Block), of 128 bytes is added for each processor. In addition, TCB (Task Control Block) is extended by 6 bytes.

TABLE I. COMPARISON OF CODE SIZE (BYTES)

	text	Data	bss
JSP	26,671	5	68
FDMP	42,707	6	76

B. Performance

First, we have measured execution times of two frequently used system calls. One is wup_tak, which wakes up a task in the wait state. We executed wup_tsk in two conditions. One condition is that the system call invokes task dispatch, and the other is that it does not. The other system call is sig_sem. Similar to wup_tsk, sig_sem was executed in the two conditions as described above. The key difference between wup_tsk and sig_sem is that wup_tsk aquires a task lock only, while sig_sem acquires both a task lock and an object lock.

Tables I and II show the results. The row labeled “JSP” presents execution times of the system calls using TOPPERS/JSP Kernel. The next row “FDMP (Intra-processor)” presents execution times in case the system calls are issued towards a task/object in the same processor using TOPPERS/FDMP Kernel. The last row “FDMP (Inter-processor)” shows the case the system calls are issued towards a task/object in a different processor. Compared with the JSP kernel, the execution times becomes longer even in case of inter-processor system calls. This is because of the additional routine for mutual exclusion and data structures being more complicated. In case of system calls with dispatch, the execution times of the FDMP (inter-processor) are longer than those of the FDMP (intra-processor). This is because of the increased overhead for dispatching a task on a different processor.

TABLE II. EXECUTION TIME OF SYSTEM CALLS WITHOUT DISPATCH

	wup_tsk	sig_sem
JSP	5 μs	5 μs
FDMP (intra-processor)	9 μs	10 μs
FDMP (inter-processor)	10 μs	10 μs

TABLE III. EXECUTION TIME OF SYSTEM CALLS WITH DISPATCH

	wup_tsk	sig_sem
JSP	7 μs	6 μs
FDMP (intra-processor)	11 μs	13 μs
FDMP (inter-processor)	17 μs	18 μs

Next, we have compared execution times of system calls which implements deadlock avoidance mechanism. The results are shown in Table IV. Compared with Table II where the system calls do not implement deadlock avoidance, the increase in execution times are large. System calls sig_sem without dispatch in Table II and rel_wai in Table IV have the similar functionality, and therefore, the performance overhead for deadlock avoidance is approximately 5 μs.

Finally, we have measured the worst-case interrupt response times. Figure 4 shows the comparison results. Method 1 is a straightforward method where interrupts are disabled first and then the lock is acquired. Method 2 is the technique used in the FDMP kernel as explained in Section III.E. We see that the interrupt response times with method 1 become long as the number of processors increases. On the other hand, our method is efficient enough to satisfy the second requirement in Section II.B.

TABLE IV. EXECUTION TIME OF SYSTEM CALLS WITH DEADLOCK AVOIDANCE

	ter_tsk	rel_wai	chg_pri
JSP	4 μs	5 μs	3 μs
FDMP (intra-processor)	11 μs	14 μs	10 μs
FDMP (inter-processor)	11 μs	15 μs	10 μs

More detailed experiments can be found in [3].

V. SUPPORTING TASK MIGRATION

In TOPPERS/FDMP Kernel, tasks are statically allocated to processors. Dynamic task migration is not supported. On one side, dynamic task migration is very effective in order to balance the loads among processors and improve average-case performance. On the other side, however, automatic task migration makes it very difficult to analyze and bound worst-case performance.

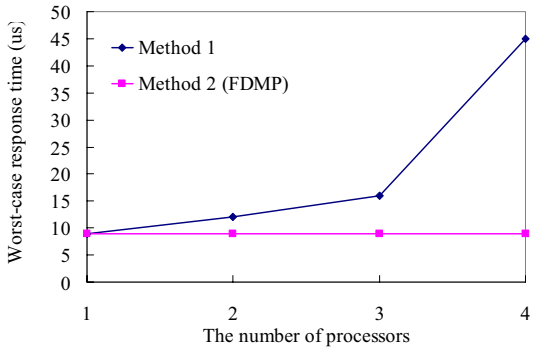

Figure 4. Worst-case interrupt response time

Based on TOPPERS/FDMP Kernel, we have developed another RTOS, named TOPPERS/FMP Kernel, which support dynamic task migration. In order not to degrade worst-case response, the FMP kernel does not automatically migrate tasks across processors. Instead, the FMP kernel provides system calls for task migration. Therefore, it is programmer's responsibility to decide when and which task is migrated to where.

VI. CONCLUSIONS

This paper discusses requirements, principles and implementation techniques for real-time operating systems to be used in real-time asymmetric multiprocessor systems-on-chip. Specifically, we present TOPPERS/FDMP Kernel and TOPPERS/FMP Kernel. The FDMP kernel is now released as open-source software from the website of TOPPERS Project [4]. The FMP kernel is at present released to members of TOPPERS Project and will be open to public in near future.

ACKNOWLEDGMENT

Implementation of TOPPERS/FDMP Kernel was in part supported by IPA.

REFERENCES

[1] TRON Association, http://www.tron.org/.

[2] H. Takada and K. Sakamura, "Inter- and intra-processor synchronizations in multiprocessor real-time kernel," *International Workshop on Parallel and Distributed Real-Time Systems*, 1996.

[3] S. Honda and H. Takada, "Extension of ITRON specification OS for function-distributed multiprocessors," *IEICE Trans. Information and Systems*, vol. J91-D, no. 4, pp. 934-944, Apr. 2008 (in Japanese).

[4] TOPPERS Project, http://www.toppers.jp/.