IEICE Electronics Express, Vol.5, No.9, 296–302

A hardware operating system kernel for multi-processor systems

Sanggyu Park[a)], **Do-sun Hong, and Soo-Ik Chae**

School of EECS, Seoul National University,

Building 104–1, Seoul National University, Gwanakgu, Seoul, 151–742, Korea

a) *sanggyu@sdgroup.snu.ac.kr*

Abstract: We propose a hardware operating system kernel (HOSK), which schedules tasks, controls semaphores, and pre-fetches contexts, as a hardware coprocessor in multiprocessor systems. A multiprocessor system can substantially reduce multithreading overheads by using the HOSK together with simplified RISC processors that do not include hardware for multithreading. We implemented an efficient HOSK which requires about 14 $\sim$ 25 K gates. The experimental results show that the multithreading overheads with a HOSK can be reduced to less than 1 percent. Preliminary efforts confirm that this approach is a feasible solution for minimizing the hardware complexity of a multiprocessor system.

Keywords: multiprocessor, RISC, operating system, multithreading

Classification: Science and engineering for electronics

References

[1] Susanna Nordström, "Application Specific Real-Time Microkernel in Hardware," *Proceedings of International Conference on Computer Designs: VLSI in computers and processors*, pp. 333–336, 2005.

[2] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai, "Hardware Implementation of a real-time operating system," *Proceedings of the 12th International Symposium on TRON Project*, pp. 34–42, 1995.

[3] P. Kohout, B. Ganesh, and B. Jacob, "Hardware Support for Real-time Operating Systems," *Proceedings of CODS-ISSS'03*, pp. 45–51, 2003.

[4] MIPS Technologies, Inc., "How to Choose a CPU Core for Multi-CPI SOC Designs," [Online] http://www.mips.com, 2002.

[5] J. Ito, T. Nakano, Y. Takeuchi, and M. Imai, "Effectiveness of a High Speed Context Switching Method Using Register Bank," *IEICE Trans. Fundamentals*, pp. 2661–2667, 1998.

[6] Kiyofumi Tanaka, "PRESTOR-1: A Processor Extending Multithreaded Architecture," *Proceeding of the Innovative Architecture for Future Generation High-Performance Processors and Systems*, pp. 91–98, 2005.

[7] Simon Segars, "The ARM9E Synthesizable Processor Family," [Online] http://www.arm.com, 1999.

DOI: 10.1587/elex.5.296
Received November 10, 2007
Accepted April 14, 2008
Published May 10, 2008

1 Introduction

When a high-performance processor that meets the performance requirements of a system is not available, many designers prefer to employ multiple processors because their performance appears to be easily scaled up by effectively integrating them. However, designing an efficient multiprocessor system is not an easy task due to multithreading overheads.

Solutions for reducing such overheads, which were originally devised for a single-processor system, are also useful for multi-processor systems. One solution is to employ a hardware accelerated operating system coprocessor that accelerates several operating system functions such as thread scheduling, semaphore control, and queue management [1, 2, 3]. However, context switching overheads cannot be reduced with the coprocessor because it cannot directly access the processor's internal registers. Another solution is to employ hardware multi-threading processors such as MIPS 32 K®processors, which support multiple contexts for fine-grain multithreading [4]. The number of contexts handled by a multithreading processor can be increased by using context caches [5, 6].

In designing multi-processor systems, however, these solutions are less attractive because they require substantial hardware overhead roughly equal to or even larger than one simple RISC processor. Therefore, instead of adopting such solutions, integrating an extra processor core could be a better choice. Furthermore, a system-level operating system is still required to suspend or wake up threads during inter-processor communications or to distribute them among multiple processors.

In this paper, we propose a hardware operating system kernel coprocessor that supports thread scheduling, inter-processor communication, and context switching in hardware for area-efficient multi-processor systems. With compact data structures and efficient hardware implementations of thread scheduling and context switching, we implemented an efficient HOSK which requires about 14 K $\sim$ 25 K gates. By using the HOSK, multithreading overhead in a multiprocessor system can be substantially reduced.

2 A HOSK coprocessor

Fig. 1 shows the block diagram of a multiprocessor system that consists of multiple simplified processors and a HOSK coprocessor comprised of three blocks: a main controller, a thread manager, and a context manager. The main controller is connected to several processors through a co-processor bus to receive system service requests. The main controller controls the other blocks in the coprocessor. The thread manager schedules threads and controls semaphore access. The context manager pre-fetches a context according to the scheduling decision of the thread manager.

2.1 Thread manager

An operating system includes a queue for ready threads. In software operating systems, a ready queue is often implemented as a linked list. However,

DOI: 10.1587/elex.5.296
Received November 10, 2007
Accepted April 14, 2008
Published May 10, 2008

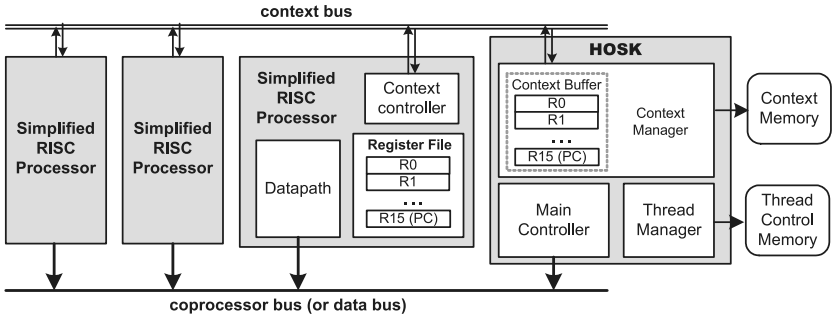

Fig. 1. A multiprocessor system with multiple simplified processors and a hardware operating system kernel

it is not suitable for simple hardware implementation because priority sorting is required for thread scheduling. For example, uTRON adopted a radix sort algorithm for task scheduling. It stores priority values of all threads in registers to complete scheduling in a few clock cycles [2], but it is not area-efficient due to the high complexity of registers and multiplexers. Although it is possible to store the priority values in more area-efficient memories such as SRAM or SDRAM, the sort algorithm requires at least $M \cdot \log_2 (N_{MAX})$ memory accesses where M is the number of ready threads and N_{MAX} is the maximum number of threads.

In multiprocessor systems using the HOSK, it is not necessary to make a thread scheduler fast enough to be performed in several clock cycles because it runs concurrently with the processors. Therefore, we substantially reduced the complexity of the HOSK by storing the priority values in either SDRAM or SRAM.

In the thread manager, we implemented a ready queue with a bit vector, where its k-th bit represents whether a thread with identifier k is ready to run or not. Other information about a thread, which is represented with a thread descriptor, is stored in the thread control memory, as shown in Figure 2. Each thread descriptor contains four fields: N, NEXT, PRIO, and WCNT. The priority of a thread is represented by two fields PRIO and WCNT, where PRIO is assigned to the more significant bits. An invalid thread is represented with a thread descriptor with PRIO value of zero. The N and NEXT fields are used to implement wait queues, which will be explained later. The bit width of PRIO is fixed to four in the current implementation and the bit widths of the WCNT and NEXT fields are $\log_2 N_{max}$, where N_{max} is the maximum number of threads supported by a HOSK coprocessor.

For task scheduling, the thread manager scans the ready queue vector to select a ready thread with the largest priority value. The PRIO field is not modified during scheduling, but the WCNT field is updated for fair scheduling. The thread manager sets the WCNT field of the selected thread to zero and it increases the WCNT fields of the other threads with the same PRIO value. If all threads in the ready queue have a positive WCNT value, the thread manager decreases the WCNT value by one for each thread to prevent

DOI: 10.1587/elex.5.296
Received November 10, 2007
Accepted April 14, 2008
Published May 10, 2008

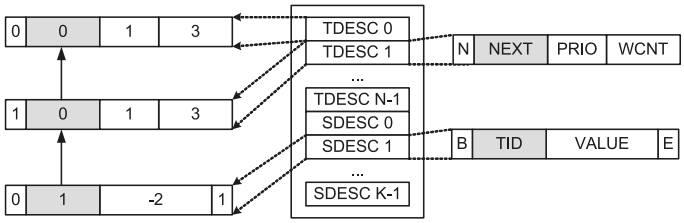

Fig. 2. Structure of the thread control memory

overflow. With this scheduling algorithm, each task scheduling requires $2 \cdot N$ memory accesses where N is the number of ready threads.

The thread manager provides inter-processor communication services that suspend or wake up threads. To make it simple, only semaphores are implemented in hardware; and mutex and interrupt services are implemented using the semaphores. For interrupt services, a semaphore is assigned to each interrupt source so that its corresponding interrupt handler thread can wait for it while a mutex is implemented as a binary semaphore with an initial value of one.

The thread control memory also stores semaphore descriptors, each of which has four fields: B, TID, VALUE, and E where B specifies whether it is a counting semaphore or a binary semaphore, and E represents its validity and VALUE means a signed semaphore value. When a thread waits for a semaphore, the thread manager decreases the field VALUE by one. Therefore, a negative value of this field means at least one thread is waiting for this semaphore.

The thread manager allocates a wait queue for each semaphore, which stores all the threads waiting for the semaphore. Note that priority sorting is not essential for the wait queue because most semaphores have only one waiting thread and the priorities of suspended threads are not changed. Therefore, we implemented the wait queue as a linked list. The TID field in a semaphore descriptor, which is valid only if the VALUE field has a negative value, specifies the thread identifier of the first entry of the wait queue. The N field of the thread descriptor is true if more entries are in the wait queue and the NEXT field specifies the thread identifier of the next suspended thread. The bit width of TDI is $\log_2 N_{max}$ and the bit width of VALUE is $(\log_2 N_{max} + 1)$.

The proposed thread manager is implemented area-efficiently in hardware because of its simplicity. If the maximum thread count is 32, the task scheduling requires 5-bit and 9-bit registers to store the identifier and priority of the highest priority thread, and one 9-bit comparator to compare priorities of threads, which corresponds to several hundreds gates. The thread control memory is also small because only 15 and 13 bits are required for each thread and each semaphore, respectively. Furthermore, the thread manager is designed to access the thread control memory with one port so that the memory can be implemented with a register file or an external SDRAM, which are more area-efficient than registers.

DOI: 10.1587/elex.5.296
Received November 10, 2007
Accepted April 14, 2008
Published May 10, 2008

2.2 Context Manager

The context manager stores the contexts of all threads in the context memory, whose size is equal to the maximum thread count times the size of a processor context. Because of its large size, the context memory should be mapped to an external SDRAM. The context manager includes a context buffer which is an on-chip memory that holds the context of the scheduled thread or the suspended thread. The context buffer is implemented with registers because it is small.

As soon as the thread scheduler completes the task scheduling, the context manager pre-fetches contexts of the scheduled thread from the context memory to the context buffer. In this time, executions of currently running threads are not interfered by the context pre-fetching. The context switching occurs when a processor core becomes idle or the priority of a pre-fetched thread is higher than a running thread in a processor core. Because the context switching is started after the context pre-fetch is completed, the SDRAM access latencies can be hidden.

For hardware-based context switching, each processor should include a context controller, which is connected to the context manager through the context bus [5]. The context controller reads the register file in the processor and writes context data to the context buffer in the context manager, and vice versa. In a processor with multiple contexts, the context controller can switch the context in the background without halting the running thread [5, 6]. In an area-efficient processor with only one context, however, the context controller should wait until all pipeline stages are emptied or completed before context switching.

Here, we assume that a simplified 5-stage pipelined processor with 16 32-bit general purpose registers and one 32-bit system register is employed. Before starting context switching, the context controller in the processor invalidates the fetch and decode stages and it waits until all issued instructions in the other stages are completed. In the processor, all instructions are completed in the execution stage except load instructions. To exploit this feature, the context controller starts context switching at the time when load/store instructions are not in the pipelines. The context manager commands the instruction and data caches to pre-fetch instruction codes and stack data so that the restored thread can be resumed without cache misses. Moreover, the context manager exchanges the program counter (PC) first so that instructions can be fetched and decoded during the context switching to reduce the context switching latency.

The context switching latency depends on the width of the context bus. If the context bus is a duplex 32-bit bus, it takes 17 $\sim$ 20 cycles. Obviously, the latency can be reduced by increasing the bit-width of the context bus. As an extreme case, it only 4 clock cycles if the context bus is a duplex 544-bit bus.

DOI: 10.1587/elex.5.296
Received November 10, 2007
Accepted April 14, 2008
Published May 10, 2008

3 Experimental Results

To evaluate the HOSK, we also implemented an experimental simplified RISC processor (REX) that adopts the ARM instruction set architecture, which has only a 3-read and 2-write register file of 16 32-bit registers. Note that for faster interrupt handling and system services, the register file in a conventional ARM9 processor has additional 15 banked-registers. In the REX, the complexity of an ALU and a load/store unit is less than 15 K gates and a context with 544 bits requires about 10 K gates. The total complexity of a REX is 25 K gates while those of fully synthesizable ARM7TDMI and ARM966ES processors are 40 K and 90 K gates, respectively [7].

Table I summarizes the area of a HOSK processor for several configurations, where the context memories are in the external SDRAM. In this experiment, the thread control memories were implemented with registers, register files and SDRAMs. Here, the maximum number of semaphores is 32 and the width of the context bus is 32 bits. When the thread control memory is implemented with SDRAM, a HOSK for 32 threads requires about 14.3 K gates, which is a little larger than the size of one context set. According to the results in Table I, we found that a HOSK for up to 128 threads can be implemented with reasonable complexity, compared to uTRON.

Table I. Area of the HOSK for different configurations

Maximum Thread count	Area (in gates including memories except SDRAM)			uTRON [2]
	Register	Register File	SDRAM	
16	22.6 K	16.5 K	13.7 K	40 K
32	26.3 K	17.8 K	14.3 K	100 K gates
64	32.7 K	20.3 K	14.8 K	190 K gates
128	45.9 K	25.1 K	15.9 K	N/A

We also compared the multithreading overheads for two cases: a REX processor with the HOSK coprocessor and an ARM9 processor with a software operating system kernel. In this experiment, a JPEG decoder is divided into four concurrent threads, each of which is executed on a different processor. Results are shown in Table II. When the context switching interval is larger than 1,000 cycles, the multi-threading overhead of the HOSK is about 1% while that of a software operating system kernel is about 32%. Moreover, the HOSK can support task scheduling for the context switching intervals of five hundred cycles with less than 4% overheads. Note that the HOSK prefetches contexts from the context memory in the external SDRAM to the

Table II. Multi-threading overheads of hardware and software operating systems

Context Switching interval (cycles)	Hardware operating system				Software operating system (ARM9 processor)
	Duplex 32-bit context bus		Duplex 544-bit context bus		
	Register file	SDRAM	Register file	SDRAM	
10,000	0.18%	0.21%	0.04%	0.1%	2.93%
1,000	1.70%	1.75%	0.41%	0.51%	32.43%
500	3.61%	3.65%	0.83%	0.88%	71.23%

DOI: 10.1587/elex.5.296
Received November 10, 2007
Accepted April 14, 2008
Published May 10, 2008

context buffer before switching the context. Therefore, the SDRAM access latencies are perfectly hidden in this experiment.

4 Conclusion

In this paper, we proposed a hardware operating system kernel which is useful for implementing area-efficient multi-processor systems. Thanks to the HOSK, multi-threaded software programs can be distributed over several simplified RISC processors with small performance overheads. To make the HOSK simple, we defined compact data structures for the thread and semaphore management and we employed a simple thread scheduling algorithm whose hardware implementation is substantially simpler relative to previous work. We think that the HOSK is also useful for hardware multi-threaded processors because it can hold multiple context sets for several processors and because it can efficiently balance dynamic work loads among the processors. To confirm this, we are now improving the HOSK to support hardware multi-threaded processors.

DOI: 10.1587/elex.5.296
Received November 10, 2007
Accepted April 14, 2008
Published May 10, 2008