Extending RTOS Functionalities: an Approach for Embedded Heterogeneous Multi-Core Systems

Shuichi Oikawa Gaku Nakagawa Naoto Ogawa Shougo Saito
Department of Computer Science
University of Tsukuba
Tsukuba, Ibaraki, Japan
{*shuioikawa,gakutarou,onaoto0707,shougosaitoh*}*@gmail.com*

***Abstract*—This paper proposes an approach to extend real-time operating system (RTOS) architecture for embedded heterogeneous muti-core processors, which consist of processors with different processing power and functionalities. The architecture splits the RTOS kernel into the two components, the proxy kernel (PK) and user-level kernel (UK). The PK runs on a less powerful core, and delegate its functions to the UK that runs on a powerful core as a user process. The experiment results running micro benchmark programs show that a communication cost between the UK and its user process is negligible and that there are cases where UK outperforms the monolithic kernel. These results confirm that the proposed approach is practically useful.**

Keywords- Real-Time Operating Systems; Heterogeneous Multi-Core Systems; Embedded Systems.

I. INTRODUCTION

As embedded devices, such as mobile smart phones, tablets, Internet TV sets, and so on, require more functions to respond to consumers' needs, their processors have been becoming more powerful. Since it is important for embedded processors to maintain their power consumption as low as possible, they cannot simply make their clock frequencies higher to increase their performance; thus, they nowadays consist of multiple processor cores and provide symmetric multi-processor (SMP) environments.

Some processors even go further and include different kinds of processor cores. The Texas Instruments OMAP4 processor [1] and the Renesas Electronics R-Mobile processor [2] are such examples. The OMAP4 processor consists of dual ARM Cortex-A9 cores and dual Cortex-M3 cores, and the R-Mobile processor consists of a Cortex-A9 core and a Renesas SH core. The OMAP4 processor incorporates Cortex-M3 cores, which are designed as microcontrollers and much smaller than but incompatible with A9 cores, to offload multimedia processing and to achieve faster real-time response. The R-Mobile processor incorporates an SH core also to offload multimedia processing. Therefore, incorporating more smaller cores to offload the specific types of processing can be a trend for future embedded processors.

Systems software, especially the operating system (OS) kernel, is, however, rather slow to respond to such an architectural change. While there have been researches conducted to support muti-core systems [3], [4], [5], they targeted server class systems with highly functional processors and their approaches do not fit into embedded processors. Since there has been no support for such embedded heterogeneous muti-core processors, only approach currently available is to execute independent OS kernels on different processors. A typical configuration for the OMAP4 processor is to execute the Linux SMP kernel on the dual Cortex-A9 and to execute an real-time OS (RTOS) on the less powerful Cortex-M3. The problems of such an architecture are 1) Linux and RTOS run independently with few cooperations between them and 2) only static functions can be provided by the software executed on the RTOS.

This paper proposes an extensible RTOS architecture for embedded heterogeneous muti-core processors. The architecture splits the RTOS kernel, which runs on a less powerful core, such as Cortex-M3 for OMAP4, and delegate its functions to the user-level kernel (UK) that runs on a powerful core, such as Cortex-A9 for OMAP4. The kernel on a less powerful core is called a proxy kernel (PK) since the global decisions are made by the UK and it works as a proxy of the UK. Figure 1 shows the overall architectures of the existing and proposed systems on OMAP4. In the figure, A9 and M3 stand for Cortex-A9 and Cortex-M3 cores of OMAP4, respectively. The architecture addresses the problems of the existing systems that consist of the independent OS kernels by making the PK closely coupled with Linux and controlled flexibly via the UK.

The rest of this paper is organized as follows. Section II describes the related work. Section III present the proposed system architecture. Section IV describes the current status and shows experiment results. Finally, Section V concludes this paper and describes our future work.

II. RELATED WORK

Recent researches conducted to support muti- or many-core systems all take basically the same approach. They consider a single system as a distributed system in order to amortize different characteristics. Multikernel [3] and Corey [4] target symmetrical processor systems with non-uniform memory access (NUMA) characteristic. By considering such

 ISBN: 978-1-61208-202-8

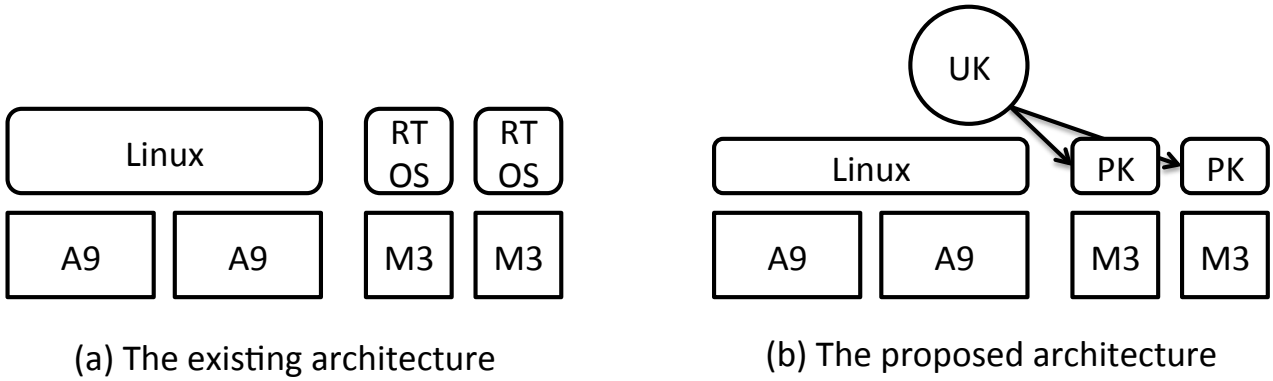

Figure 1. Overall architectures of the existing and proposed systems.

systems as distributed systems, they can partition systems into clusters of processors with the same characteristic, and can hide NUMA characteristic. On the other hand, the architecture proposed in this paper targets heterogeneous multi-core systems, of which processors have different functionalities.

Helios [5] targets heterogeneous multiprocessor systems and supports different kinds of processors by employing a satellite kernel, which is a microkernel [6]. Each satellite kernel on a different processor exports the same API, so that programs can be executed on a processor that fit the programs' requirements. Although the architecture proposed in this paper targets heterogeneous multi-core systems, which is similar to the target of Helios, the kernels on different kinds of processors are not the same. It aims to make programs on a less powerful core closely coupled with Linux on a powerful core by complementing the functionalities of the small kernel on a less powerful core.

III. Proposed System Architecture

This section describes the proposed RTOS architecture and its components in detail. As depicted in Figure 1, the proposed architecture consists of three major components, the PK (Proxy Kernel), the UK (User-Level Kernel), and Linux. The PK and the UK constitutes the RTOS kernel. The PK is executed on a less powerful core, and its functionality is supported by the UK that is executed on a powerful core as a user process of Linux.

A. PK: Proxy Kernel

The PK is a simplified RTOS kernel. It is a standalone kernel; thus, it consists of basic RTOS components, such as interrupt and exception handlers, a scheduler, and synchronization mechanisms. It works with the UK so that its functions are complemented by the UK. It does not perform dynamic resource management except for task scheduling. It simply picks up the highest priority task and dispatch it. It processes interrupts, and unblocks tasks when needed. When a fixed task set is executed on it, it works the same way as an ordinary RTOS.

The PK works in different ways from an ordinary RTOS when dynamic management features are involved. It outsources such features to the UK, so that it can keep itself as simple as possible while its functionality can be extensible. When a task on the PK invokes a system call the PK itself cannot handle, the system call is transferred to and processed by the UK on behalf of the PK. Some exceptions are processed in the same way. By outsourcing the functions to the UK, the OS functionalities provided for tasks on the PK become flexible and extensible. For example, file access and networking features can be easily provided through the UK since it is executed on Linux. Moreover, the PK outsources its memory management to the UK since it is unnecessary to execute a fixed task set. It is possible because the physical memory of the PK is mapped into the UK's address space. The task management, especially the creation and deletion of tasks, uses the memory management functions. Thus, creating a new task is a function of the UK, and the PK simply dispatches it when it becomes ready to run. When a task exits, such an event is transferred to the UK, and the memory used by the existed task is reclaimed by the UK.

B. UK: User-Level Kernel

The UK processes the requests issued by the PK's tasks on behalf of the PK as described above. The UK is executed as a user process of Linux; thus, it can utilize the full functionalities of Linux. It can access files on Linux's file systems and operate on networks just as Linux's user processes can. It can provide the PK with such Linux's functionalities without increasing the complexity of the PK, and can easily introduce dynamic features to the PK.

The other benefit for the UK to be a user process of Linux is that it is free from maintaining its own execution environment, and can focus on managing the execution environments of the PK. Such delegation of functions makes the implementation of each components as simple as possible.

 ISBN: 978-1-61208-202-8

Finally, there must be a communication means between the PK and the UK. As a user process of Linux, the UK cannot directly communicate with the PK without the support from Linux. There are two ways for the UK to communicate with the PK. One is through the read and write system calls, and the other is through the shared memory. The first method is simpler while it involves overheads of using system calls. The latter is faster while it complicates the interactions between the UK and the PK. Both methods need to be considered for the better implementation.

IV. Current Status and Experiment Results

This section describes the current status, experiment results, and a performance improvement based on profiling. The PK and UK were implemented based on the XV6 operating system [7], which is a reimplementation of UNIX V6 [8]. The current implementation is based on an Intel IA-32 multi-core processor because we could not obtain an OMAP4 based system when we started the work. It statically considers some cores as powerful ones and some as less powerful ones. Therefore, all experiments described below were performed on a PC-AT compatible system, which is equipped with an Intel Core i7-920 2.66GHz CPU. The hyper threading and power management features were disabled to perform all benchmarks. We used the Scientific Linux 6.1 x86_64, which is based on the Linux kernel 2.6.32, to execute the UK. While the Linux kernel executes in the 64-bit mode, the UK is a 32-bit program. The original XV6, which is used for comparisons, executes in the 32-bit mode.

A. Current Status

The implementation consists of 3 parts, the PK, the UK, and the linux device driver to interact with the PK, as described in Section III. The PK consists of total 1534 lines, which are 1448 lines of the C program and 86 lines of the assembly program. The linux device driver consists of total 830 lines, which are 723 lines of the C program and 107 lines of the assembly program. As far as the UK is concerned, 12 files, mostly for device drivers, were deleted, 5 files were added, and 13 files were modified from the original XV6. The total number of lines of the added files are 494 lines.

The current implementation is stable enough to perform micro benchmark programs as follows.

B. Micro Benchmarks

We first executed several micro benchmark programs on the UK and also on the original XV6 in order to investigate the performance penalty to realize the proposed architecture. We chose 4 programs, getpid, pipe, fork, and fork+exec to measure the functions without and with dynamic resource management. The getpid program invokes the getpid system call to find the cost of calling the kernel. The pipe program makes 2 processes communicate with each other through 2 pipes, so that it can measure the cost of context switches between them. The fork program creates a copy of the current process, and the fork+exec program executes a different program in a newly created process. These programs can measure the process management costs.

Table I
Micro Benchmark Results [in μsec]

Benchmark	UK	UK (mmap buf)	Original XV6
getpid	1.06	0.68	0.20
pipe	10.47	8.98	2.77
fork	23.80	22.05	82.63
fork+exec	49.99	48.17	168.56

Table I shows the results of executing micro benchmark programs. In the table, UK uses the linux device driver to communicate with the PK. On the other hand, UK (mmap buf) directly communicates with the PK through a buffer that is mapped in the UK and the PK's address spaces; thus, it does not use the linux device driver to communicate with the PK. The original XV6 was executed directly on a system.

The results from the getpid and pipe programs show the overheads incurred by splitting the execution of the UK and its user processes on different processors. Since the getpid program only invokes the getpid system call, the difference between the results of UK and XV6 is the communication cost between them; thus, the communication cost is 0.86 μsec for UK and 0.48 μsec for UK (mmap buf). Direct communication through the mapped buffer without the linux device reduces the communication cost by 44%. While the overall cost for UK (mmap buf) to invoke the getpid system call increases as much as 3.4 times more than XV6, the cost increases by only 0.48 μsec, which is negligible in the total computing time of applications and other more complicated system calls. Moreover, such simple system calls as getpid, which obtains the state of in-kernel resources but does not manipulate them, can be optimized by embedding them within the PK. In this case, the communication costs are eliminated, and the costs to invoke such simple system calls becomes the same as XV6.

The results from the fork and fork+exec programs show that UK performs better than XV6. This is an advantage of the proposed architecture. The control flows to process the fork and exec system calls currently remain the same for UK and XV6 since no optimization, such as batching multiple system calls and parallelizing the execution of the kernel and user processes, has been applied to UK. We consider the differences arise due to the effects of cache and TLB. There is no need for the UK to flush TLB of the processor it is running since actual process manipulation is done on another processor user processes are running. The PK runs on the processor that runs user processes. It is however extremely small; thus, the effect to it is

 ISBN: 978-1-61208-202-8

Table II
PROFILING RESULTS (TOP 5)

Function Name	%
memmove	35.48
jpkwaitevent	11.93
memset	9.18
freevm	6.92
getcallerpcs	3.78

negligible. Moreover, having the UK and user processes run on different processors increases the chances for them to remain on cache. Therefore, the UK architecture can decrease the number of cache and TLB flushes and increase the performance.

C. Profiling Results and Improvement

In order to further improve the performance of UK, we analyzed the hot spots in the UK. We used OProfile, which is a system wide profiler for Linux systems. Profiling was taken while running the fork and fork+exec programs. Table II shows the results of profiling. The table only shows the top 5 function names where the most of CPU cycles were consumed. These top 5 functions consumes 67.29% of CPU cycles in total.

The function that most consumes CPU cycles is memmove, which copies data from one place to the other. The second one is jpkwaitevent, which busy waits the completion of the user process side processing; thus, it does nothing. The third one is memset, which sets a memory region to a specified value. The two string functions, memmove and memset, consumes 44.66% of CPU cycles in total; thus, their performance should impact the overall performance.

Intel Core-i7 supports SSE4, which is a SIMD unit that has 8 128-bit long registers. Since a SIMD unit similar to Intel SSE is also available for the ARM architecture as NEON, we decided to utilize it to accelerate memory operations. A single SSE instruction can move 128-bit (16-byte) data between a SSE register and memory. We utilized a SIMD instruction to accelerate memmove and memset. By using the SIMD versions of them, the performances of the fork and fork+exec benchmark programs were improved 27% and 13%, respectively.

Figure 2 summarizes the results from the performed micro benchmark programs.

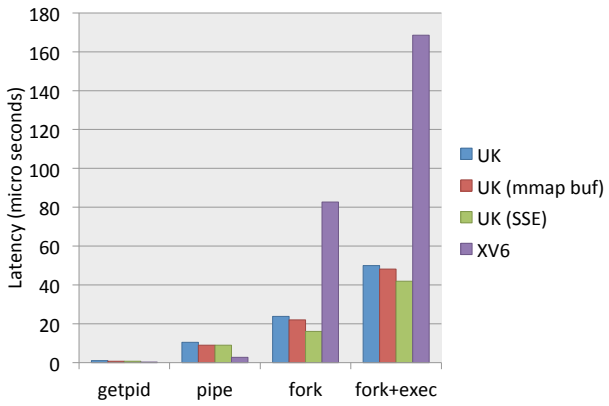

Figure 2. Summary of Micro Benchmark Results

V. SUMMARY AND FUTURE WORK

This paper proposed an extensible ROTS architecture for embedded heterogeneous muti-core processors, which consist of processors with different processing power and functionalities. The architecture splits the RTOS kernel into the two components, the PK and UK. The PK runs on a less powerful core, and delegate its functions to the UK that runs on a powerful core as a user process. The experiment results running micro benchmark programs show that a communication cost between the UK and its user process is negligible and that there are cases where UK outperforms the monolithic kernel. We now obtained an OMAP4 based evaluation board [9], and are currently porting the proposed architecture on it.

REFERENCES

[1] D. Witt. OMAP4430 Architecture and Development. Hot Chips Symposium, August 2009.

[2] M. Ito, et. al. SH-Mobile G1: A Single-Chip Application and Dual-mode Baseband Processor. Hot Chips Symposium, October 2006.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: a New OS Architecture for Scalable Multicore Systems. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, pp. 29-44, October 2009.

[4] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an Operating System for Many Cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pp. 43-57, December 2008.

[5] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, pp. 221-234, October 2009.

[6] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an Application Program. In *Proceeding of the USENIX Summer Conference*, pp. 87-95, June 1990.

[7] Xv6, a Simple Unix-like Teaching Operating System. http://pdos.csail.mit.edu/6.828/xv6/

[8] J. Lion. Lion's Commentary on UNIX V6.

[9] Pandaboard. http://pandaboard.org/ (as of 3 April 2012).

 ISBN: 978-1-61208-202-8