Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2010, Article ID 458085, 8 pages
doi:10.1155/2010/458085

Research Article

Composition Kernel: A Software Solution for Constructing a Multi-OS Embedded System

Yuki Kinebuchi, Kazuo Makijima, Takushi Morita, Alexandre Courbot, and Tatsuo Nakajima

Department of Computer Science, Waseda University, 3-4-1 Okubo Shinjuku-ku, Tokyo 169-8555, Japan

Correspondence should be addressed to Yuki Kinebuchi, yukikine@dcl.info.waseda.ac.jp

Received 2 December 2009; Accepted 12 October 2010

Academic Editor: Chun Jason Xue

Modern high-end embedded systems require both predictable real-time scheduling and high-level abstraction interface to their OS kernels. Since these features are difficult to be balanced by a single OS, some methods that accommodate multiple different versions of OS kernels, typically real-time OS and general purpose OS, into a single device have been proposed. The hybrid kernel, one of those methods, executes a general purpose OS kernel as a task of real-time OS which can support those features with reasonable engineering effort. However when adapting the approach to various combinations of OS kernels, which is required in the real-world embedded system design, the engineering effort of modifying the kernel becomes not negligible. This article introduce a method called a *composition kernel* which uses a thin abstraction layer for accommodating kernels without making direct dependencies between them. The authors developed the abstraction layer on an SH-4A processor and executed kernels on top of it. The amount of modifications to the kernels was significantly smaller than that in related work, while introducing only negligible verhead to the performance of the kernels.

1. Introduction

One of the difficulties of designing an OS kernel for embedded systems is supporting both predictable real-time scheduling and high-level abstraction interface. For instance, modern cell-phones need to execute both real-time control software (such as a radio transmitter device controller) and function-rich applications (such as a web browser, a media player). Traditionally, small real-time operating systems (RTOSs), that supports predictable real-time scheduling, were used for embedded system software development. Today, along with the expansion in the variety of functions provided by a single embedded device, general purpose operating systems (GPOSs) are widely used in modern embedded system development. GPOS offers a high-level abstraction interface which eases the development of function-rich applications. However, with GPOS kernels, it is hard to provide predictable scheduling because their large code base makes the activities in the kernel indeterministic. Various studies indicates that a lot of engineering effort is required for extending function-rich kernels to support predictable real-time scheduling [1, 2].

Instead of putting effort in extending a large and complex kernel to support predictable scheduling, there are relatively straightforward approaches to use both RTOS and GPOS kernels in a single device. One is a hardware-based approach which assigns dedicated hardware set to each OS. For instance, modern mobile phones use a MultiProcessor System on Chip (MPSoC) to assign a dedicated processor to each OS kernel [3]. Another approach uses a hypervisor, or a virtual machine monitor (VMM), which multiplexes a physical device into multiple virtual devices that each of them is capable of executing an OS. Although virtualization technologies for embedded systems have attracted attentions these past years, they are not widely adopted by real-world products because only few embedded processors support virtualization extensions.

A hybrid kernel is an OS kernel architecture which executes a GPOS kernel on top of a RTOS kernel [4–6]. The GPOS kernel runs as a task of the underlying RTOS kernel.

Some previous studies have proven that the hybrid kernel is applicable to use in real-world embedded systems [7, 8]. Hybrid kernel balances predictable real-time scheduling and highly abstracted interface. Furthermore, it does not rely on a specific MPSoC. A disadvantage of hybrid kernel design is that the GPOS kernel depends on the interface of the underlying RTOS kernel. Typically embedded device manufacturers use diverse RTOS kernels depending on real-time constraints, software properties they own, and so forth. Although the engineering cost of modifying a GPOS kernel to a RTOS task is reasonably small, the engineering cost to support various combinations of RTOS kernels and GPOS kernels will introduce significant engineering effort.

This paper proposes a method called a *composition kernel* which enables to constructs a multiOS-kernel system with minimal engineering effort and without sacrificing the performance of the OS kernels. The method is based on a thin abstraction layer which multiplexes a processor. The OS kernels do not depend on each other but on the underlying abstraction layer which exposes an interface almost identical to the real processor instructions. Thus, it requires less engineering effort than the hybrid kernel and introduces only negligible overhead. The abstraction layer ease the developments of multiOS-kernel systems with various combinations of RTOSs and GPOSs.

The contributions of this paper are proposing the composition kernel method, and showing its validity by implementing and evaluating it using real-world software and hardware. We developed an abstraction layer named SPUMONE from scratch to run on the SH-4A processor architecture [9]. SPUMONE can execute the TOPPERS/JSP [10] RTOS (TOPPERS/JSP is a RTOS kernel with the μITRON specification, which is an OS interface specification for embedded systems widely used in the Japanese industry,) the OKL4 microkernel [11] and the Linux kernel on top of it. Its design and implementation is simple and efficient to accommodate multiple OSs together with a few dozen lines of modifications to both OS kernels while maintaining the real-time responsiveness of the RTOS. The remainder of the paper is organized as follows. Section 2 introduces some related work on system designs using multiple OS kernels and compare them with our contributions. Section 3 describes the design and the implementation of our method. Section 4 shows the results of the evaluations showing that our method requires small engineering effort and offers low overhead. Section 5 discusses the necessity of strong isolation for embedded multiOS-based system. Finally Section 6 concludes this paper.

2. Related Work

Various approaches are proposed to balance real-time responsiveness and high-level abstraction on a single device. One of the approaches is modifying a GPOS kernel to support real-time responsiveness [2, 12]. The real-time patch extends Linux to support kernel preemption [12]. It achieves a few hundred μseconds latency [13], but still the result is slower by a factor of ten comparing to typical RTOSs. Although these approaches are potentially capable of supporting real-time response time, it is challenging to maintain their response time through continuous revisions of the OSs. In addition, porting all the software from the RTOS to Linux would impose substantial engineering cost.

Instead of supporting both real-time responsiveness and high-level abstraction interface by a single version of an OS kernel, modern embedded systems use multiple versions of OS kernels: typically a RTOS kernel and a GPOS kernel in one device. The most simple approach of accommodating multiple kernels is assigning a dedicated set of hardware to each OS kernel. For instance, by using MPSoC, a dedicated processor core and memory is assigned to each OS kernel. Hardware-based multiOS-kernel design is straightforward, however the cost of developing a MPSoC is not flexible compared to software-based approach. The decision of adopting hardware or software depends on constraints, therefore cannot say which is better. In this paper we focus on software-based multiOS-kernel designs.

Another approach is using a hypervisor, or a virtual machine monitor (VMM). It is a similar idea to the MPSoC based approach, but the underlying hardware is virtually multiplexed by software. It can accommodate RTOS and GPOS into a single embedded device without any modifications or with just minimal modifications to the OS kernels [14–17]. Hypervisor's design is broadly classified into full virtualization and paravirtualization. A hypervisor that supports full virtualization exposes a virtual hardware interface identical to a real hardware interface. OSs can be executed without any modification on full virtualization. On the other hand, implementing full virtualization complicates the design of the hypervisor itself or requires hardware support [18]. Unfortunately the hardware virtualization support is still an unfamiliar feature for embedded system processors. This motivates embedded system hypervisors to use paravirtualization for their system design [19–21]. In this case, the engineering cost required for paravirtualizing a guest OS kernel is problematic for manufacturers. In addition, switching the privilege level between a guest OS and a hypervisor will entail performance degradation. Furthermore the OS isolation can spoil the real-time responsiveness and system throughput.

Another approach is the hybrid kernel, a GPOS kernel built on top of a RTOS kernel. RTLinux [4] replaces Linux kernel's hardware abstraction layer (HAL) with its own version of a microkernel (or a RTOS kernel). The microkernel is executed in privileged mode together with the Linux kernel. Its interrupt response time is a few μseconds, which is comparable to typical RTOSs. However the microkernel exposes an specific programming interface, which prevents reusing real-time applications developed for other RTOSs. In contrast, some hybrid kernels use existing RTOS kernels for their microkernel layers. Linux on ITRON replaces the Linux HAL with an existing μITRON compatible RTOS [6]. This architecture enables the system to reuse both the software developed for Linux and the μITRON RTOS specification. The hybrid kernel provides high real-time responsiveness comparable to an traditional RTOS along with exposing high-level abstraction interface by reusing existing GPOS kernels.

The problem of hybrid kernels is the dependencies between a GPOS kernel and a RTOS kernel. The GPOS kernel depends on the interface exposed by the underlying RTOS kernel. Typically embedded device manufacturers use diverse RTOS kernels depending on real-time constraints, software properties they own, applications supported, and so froth. Even though the engineering cost of modifying a GPOS kernel to run as a RTOS task is reasonably small, the engineering cost for supporting various combinations of RTOS kernels and GPOS kernels is problematic. Adeos [22] and Xenomai [23] provide a low-level abstraction interface for building RTOS compatible layer, which is similar to a composition kernel. However, Adeos depends on the 4 ring levels supported by the Intel processor architecture, which cannot be applied to the other typical embedded processor architectures supporting only 2 privilege levels. Xenomai exposes an abstraction interface in the user level of Linux, whose purpose is to support a programming environment compatible with existing RTOSs. Its goal is to support the transparent porting of RTOS applications but not RTOS kernels.

The composition kernel method can execute an OS kernel on top of it with minimal engineering effort, which gives a flexibility of accommodating various combinations of OS kernels in a single device. In order not to penalize performance, our abstraction layer executes the OS kernels as well as itself in the same privileged level. This also reduces the engineering cost of modifying OS kernels, because a majority of privileged instructions can be executed directly by a processor and only a minimal set of instructions needs to be emulated. Furthermore, The abstraction layer multiplexes only minimal hardware resources. The other resources are exclusively assigned to each OS by simply reconfiguring each OS kernel not to access the same devices.

3. Design and Implementation

This section introduces a composition kernel, a method for constructing an embedded device with multiple OSs. The method is based on a simple abstraction layer called SPUMONE and some modifications to OS kernels.

3.1. SPUMONE. SPUMONE (Software Processing Unit, Multiplexing ONE into two or more) is a thin software layer for multiplexing a single physical processor into multiple virtual ones. Unlike hypervisors, SPUMONE itself and OS kernels are executed in privileged mode as shown in Figure 1. If an OS does not support user land, its applications would be executed in privileged mode altogether.

This contributes to minimize the overhead and the amount of modifications to the OS kernels. Furthermore it makes the implementation of SPUMONE itself simple. Executing OS kernels in nonprivileged mode complicates the implementation of the abstraction layer, because various privileged instructions have to be emulated. The majority of the kernel and application instructions, including the privileged instructions, are executed directly by the real processor, and only the minimal instructions are emulated by SPUMONE. These emulated instructions are invoked from the OS kernels in a simple function call; it eliminates the overhead of trapping between privileged and nonprivileged mode for system-calls and hypercalls.

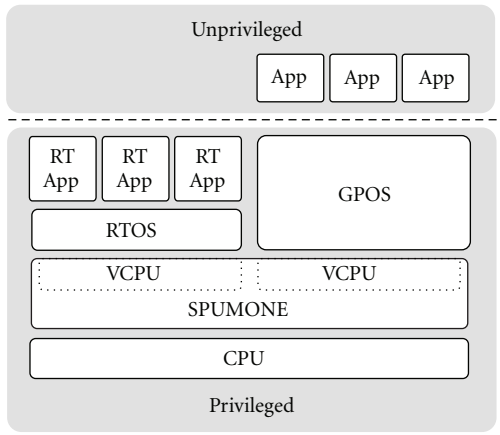

Figure 1: SPUMONE based system on a single-core processor.

Since the interface has no binary compatibility with the original processor interface, we simply modify the source code of OS kernels, a method known as paravirtualization [15, 17]. Thus we assume we have access to the source code of the OS kernels. The modifications required to the OS kernels are described in detail in Section 3.2.

3.1.1. Virtual Processor Scheduling. A processor is multiplexed by scheduling the execution of OS kernels. The execution states of the OSs are managed by a data structure that we call a virtual processor. When switching the execution of the virtual processors, all the hardware registers are stored into the corresponding virtual processor's register table, and then loaded from the table of the virtual processor executing next. The mechanism is similar to the process paradigm of a classical OS, but in addition it also saves the privileged control registers, that is, the entire processor state.

The scheduling algorithm of virtual processors is a fixed priority preemptive scheduling. A virtual processor bound to the RTOS will gain a higher priority than a virtual processor bound to the GPOS in order to maintain the real-time responsiveness. This means the GPOS is executed only when the virtual processor for the RTOS is in an idle state and has no task to execute. The process or task scheduling is left up to an OS kernel so the scheduling model for each OS is maintained as is. The idle RTOS resumes its execution when it receives an interrupt. The interrupt for RTOS preempts the GPOS immediately, even when the GPOS is disabling interrupts.

3.1.2. Interrupt/Trap Delivery. Interrupt virtualization is a key feature of SPUMONE. It investigates interrupts before delivering them to each OS. When SPUMONE receives an interrupt, it looks up the interrupt destination table to see to which OS the interrupt should be sent. The assignment of interrupt sources and OSs is statically defined. Traps are also sent to SPUMONE first, then are directly forwarded to the currently executing OS. To let SPUMONE receive interrupts

before the OS kernels, we modified the entry point of the interrupts to SPUMONE's vector table. The entry point of each OS is notified to SPUMONE via a virtual instruction for registering their vector table.

When the interrupt triggers an OS switch, all the registers of the current OS are saved into the register stack, then the register stack for the other OS is loaded. Finally the execution branches into the entry point of the destination OS. The processor registers are setup just as if the real interrupt occurred, so the code of the OS kernel's entry points do not need to be modified.

The interrupt enable and disable instructions are also replaced by the virtual instruction interface. Typically OS disables all interrupt sources when entering a critical section. In our approach, by leveraging the interrupt priority leveling (IPL) mechanism of the processor, we assign the higher half of the interrupt priority levels to the RTOS and the lower half to the GPOS. When the GPOS tries to block the interrupts, it modifies its interrupt mask to the middle priority. The RTOS may therefore preempt the GPOS even when disabling the interrupts (Figure 2(1)). On the other hand when the RTOS is running, the interrupts are blocked by the processor (Figure 2(2)). These blocked interrupts could be sent immediately when the GPOS is dispatched.

3.2. OS Kernel Modifications. The following describes the points of the OSs to be modified in order to run on top of SPUMONE.

3.2.1. Interrupt Vector Table Register Instruction. The instruction registering the address of a vector table is modified to notify the address to SPUMONE's interrupt manager. Typically this instruction is invoked once during the OS initialization.

3.2.2. Interrupt Enable and Disable Instructions. The instructions enabling and disabling interrupts are typically provided as kernel internal APIs that are typically coded as inline functions or macros in the kernel source code. For the GPOS, we replace those APIs with the instructions enabling the entire level of interrupts and disabling only low priority interrupts. For the RTOS, we replace those APIs with the instructions enabling only high priority interrupts and disabling the entire level of interrupts. Therefore, interrupts assigned to the RTOS are immediately delivered to the RTOS, and the interrupts assigned to the GPOS are blocked during the RTOS execution.

Figure 3 shows the interrupt priority level (IPL) assignment for each OS, which we used in the evaluation environment.

3.2.3. Physical Memory. A fixed physical memory area is assigned to each OS. The physical address for the guest OSs can be simply changed by modifying the configuration file or their source code. Virtualizing the physical memory would impose a large code into the virtualization layer and substantial performance overhead. In addition, unlike the virtual machine monitor for enterprise systems, embedded systems have a fixed number of OSs. According to these reasons we assigned fixed physical memory area for each OS.

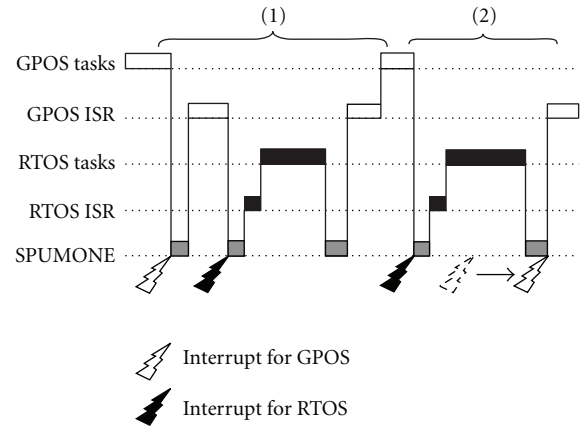

Figure 2: Interrupt delivery mechanism.

3.2.4. Idle Instruction. On a real processor, an idle instruction suspends a processor till it receives an interrupt. On a virtualized environment, this is used to yield the use of real processor to another guest OS. We prevent the execution of this instruction by replacing it with the SPUMONE API. Typically this instruction is embedded in a specific part of kernel, which is fairly easy to find.

3.2.5. Peripheral Devices. Peripheral device assignments are done by modifying the configuration of each OS in such a way that it does not share the same peripherals. The embedded multiOS-kernel system is designed to use different OSs for managing different devices. For instance, an RTOS is used for controlling specific peripherals such as a radio transmitter and some digital signal processors, and a GPOS for controlling a display and buttons.

However some devices cannot be assigned exclusively to each OS because both systems need to use them. For instance, an interrupt controller need to be shared. Usually the OS clears some I/O registers during its initialization. In the case of running on SPUMONE, the OS booting after the first one should be careful not to clear or overwrite the settings of the OS executed first. We modified the Linux initialization code to preserve the settings done by TOPPERS.

4. Evaluation

We evaluated the engineering cost of modifying the guest OS kernels, the basic overhead introduced to the OSs running on SPUMONE, and finally the real-time responsiveness of an RTOS running on SPUMONE. The evaluation is done on the SH-2007 reference board, with the SH-4A 400 MHz processor and 128 MB memory. We use TOPPERS/JSP 1.3 as RTOS and Linux 2.6.24.3 as GPOS. Linux is configured to mount an NFS share exported by the host machine and a compact flash card formatted in ext2.

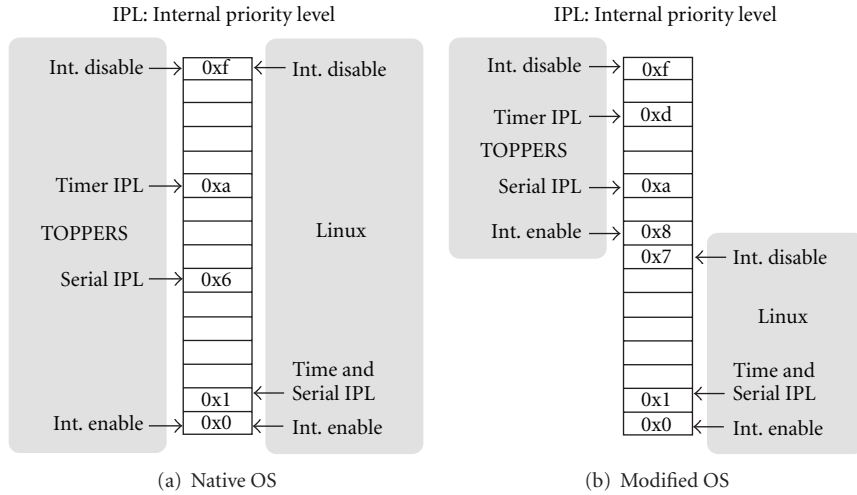

Figure 3: The interrupt priority levels assignment.

4.1. Engineering Cost. The first evaluation is the engineering cost of reusing the RTOS and the GPOS by comparing the number of modified lines of code (LOC) in each guest OS kernel. Table 1 is a list of the modified files in Linux. Table 2 shows the amount of code added and removed from the original OS kernels. Since we could not find RTLinux, RTAI, OK Linux for the SH-4A processor architecture, we evaluated them developed for the x86 architecture. OK Linux is a Linux kernel virtualized to run on the L4 microkernel. For OK Linux, we only counted the code added to the architecture dependent directory `arch/l4` and `include/asm-l4`. The comparison would not be fair in a precise sense, however as the table shows, it is clear that our approach requires significantly small modifications to the Linux kernel. This result is achieved thanks to executing guest OS in privileged mode.

4.2. Basic Overhead. For evaluating the basic overhead of SPUMONE, we measured the overhead of interrupt handling delay, and the time to build the Linux kernel on top of native (an unmodified OS running on bare-metal hardware) Linux and modified Linux, respectively. Table 3 shows the average and the worst case CPU cycles spent to handle the interrupts sent to native TOPPERS and modified TOPPERS. In the average case SPUMONE imposes 0.67 μs overhead to the delay. The worst case overhead shows the time required to save the state of Linux and restore the state of TOPPERS. The increased delay is sufficiently small and predictable for executing real-time applications.

Table 4 shows the time required to build Linux kernel on native Linux and modified Linux executed on top of SPUMONE together with TOPPERS. TOPPERS only receives the timer interrupts every 1 ms, and executes no other task. The result shows that SPUMONE and TOPPERS impose an overhead of 1.4% to Linux performance. Note that the overhead includes the cycles consumed by TOPPERS. The result shows that the overhead of the virtualization to the system throughput is sufficiently small.

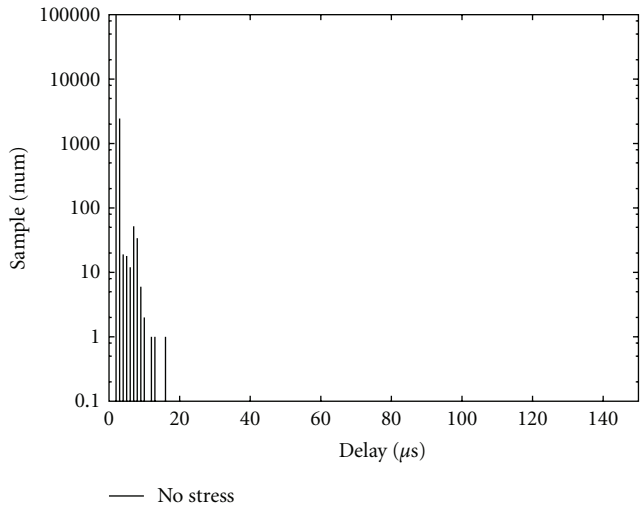

Figure 4: Dispatch delay (no `stress` on Linux without IPL modification).

4.3. Effect of Linux Load to TOPPERS Real-Time Properties. We measured how activities on Linux affects the dispatch delays of a TOPPERS periodic task in the experimental multiOS-kernel environment. The periodic task runs every 1 ms. The delays are sampled 100,000 times during the measurement. A dispatch delay is the time spent between the hardware interrupt trigger and the beginning of the periodic task. TOPPERS executes only the periodic task; there are no other TOPPERS tasks that affect dispatch delays of the periodic task. Delays were fixed to 2 μs when executing TOPPERS by itself.

We measured delays without and with applying the IPL assignment (described in Section 3.2, Figure 3) to the Linux kernel, while executing `stress` [24] on Linux with different options. In the graphs (Figures 4–11), y-axes show the number of times in the log scale that the task executed

Table 1: A list of the modifications to the Linux kernel.

File	Function/variable	Description
.config	`CONFIG_MEMORY_START` `CONFIG_MEMORY_SIZE`	Modified to use the upper half (64 MB) of the main memory
setup.c	`sh2007_setup(char **cmdline_p)`	Modified not to overwrite the value in the interrupt controller register set by TOPPERS
setup-sh7780.c	`intc2_irq_table`	The interrupt source table. Removed one of the serial devices which is used by TOPPERS
head.S	Flag register initial value	Modified IPL, not to block the interrupts for TOPPERS
traps.c	`per_cpu_trap_init(void)`	Replaced the vector table register instruction with SPUMONE API

irqflags.h	`raw_local_irq_disable(void)`
	`__raw_local_irq_disable(void)`
	`raw_local_irq_restore(void)`
		Modified not to mask the interrupts assigned to TOPPERS

processor.h	`cpu_sleep()`	Replaced the idle instruction with the SPUMONE API

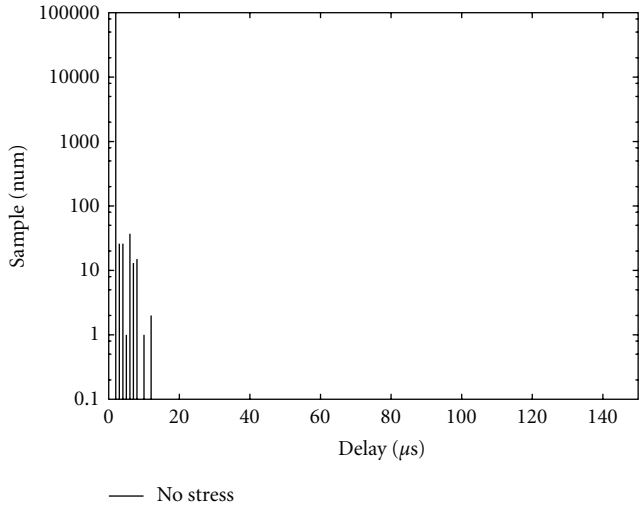

Figure 5: Dispatch delay (no `stress` on Linux with IPL modification).

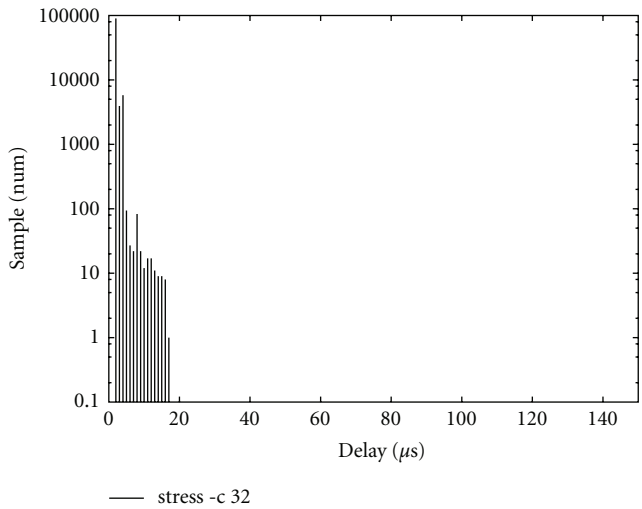

Figure 6: Dispatch delay (CPU `stress` on Linux without IPL modification).

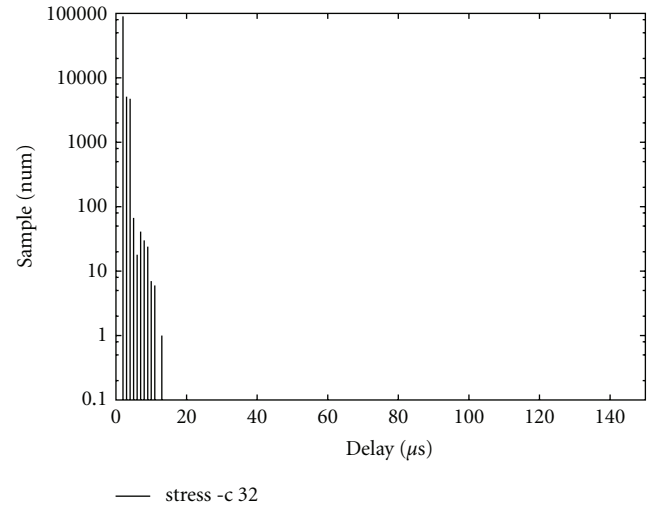

Figure 7: Dispatch delay (CPU `stress` on Linux with IPL modification).

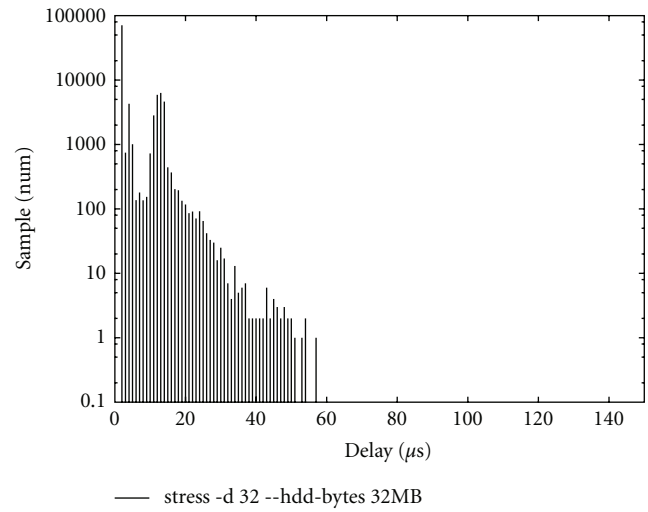

Figure 8: Dispatch delay (NFS read/write `stress` on Linux without IPL modification).

Table 2: The total number of modified LOC in *.c, *.S, *.h, Makefiles.

OS	Added LOC	Removed LOC
Linux on SPUMONE (Linux 2.6.24.3)	161	8
RTLinux 3.2 (Linux 2.6.9)	2798	1131
RTAI 3.6.2 (Linux 2.6.19)	5920	163
OK Linux (Linux 2.6.24)	28149	—

Table 3: The delay of handling the timer interrupts in TOPPERS. Over 20,000 interrupts were measured to obtained the average and the worst case time.

Configuration		CPU Clocks	Time (μs)	Overhead (μs)
TOPPERS (native)	average	102	0.25	—
	worst	102	0.26	—
TOPPERS	average	367	0.92	0.67
on SPUMONE	worst	1582	3.96	3.70

Table 4: Linux kernel build time.

Configuration	Time	Overhead
Linux only	68 m5.898 s	—
Linux and TOPPERS on SPUMONE	69 m3.091 s	1.4%

with the delay at x-axes. Figures 4, 6, 8 and 10 show results using Linux without the IPL assignment. Figures 5, 7, 9 and 11 show results using Linux with the IPL assignment. Figures 4 and 5 show the delays while Linux is unloaded. Figures 6 and 7 show the delays while Linux is loaded with CPU bound operations (`stress -c 32`). Figures 8 and 9 show the delays while Linux is loaded with the read and write operations against the NFS share (`stress -d 32 --hdd-bytes 32MB`). Figures 10 and 11 show the delays while Linux is loaded with the read and write operations against the CF card file system (`stress -d 32 --hdd-bytes 32MB`).

The measurements with the CPU bound operations show similar results with and without the IPL assignment. The measurement with the CF card operations and without the IPL assignment, shows maximum delay of 111 μs. With the IPL assignment, the maximum delay is reduced to 34 μs. Comparing these results to the measurements done by [13], with a 1.8 GHz Athlon processor which shows the maximum delay of a few hundred μ seconds, we can see that our measurements with 400 MHz SH processor achieves fairly small dispatch delays.

5. Do We Need Strong Isolation for Embedded Multi-OS-Based System?

Strong isolation among guest OSs is an attractive feature for constructing a secure and reliable embedded system [20]. However, unlike the VMMs used in the area of enterprise systems, most embedded systems consist of a fixed number of OSs. In addition, because the guest OSs are statically decided by the hardware manufacturer, they can be assumed

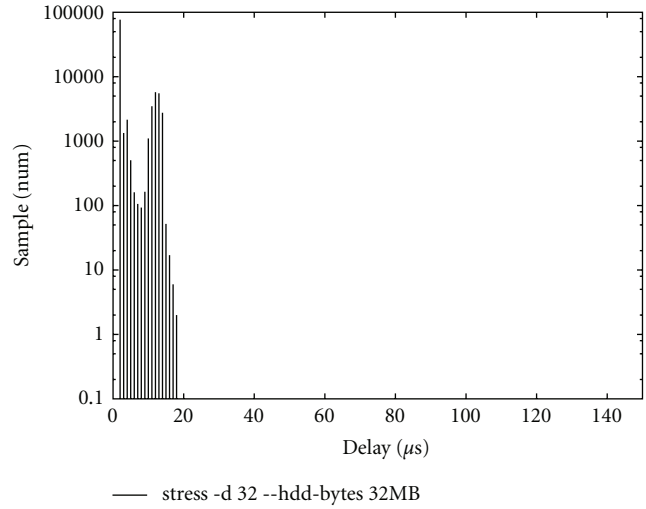

Figure 9: Dispatch delay (NFS read/write `stress` on Linux with IPL modification).

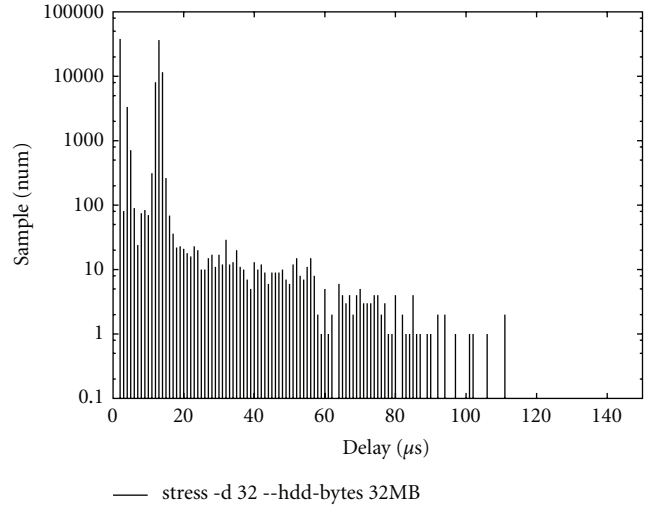

Figure 10: Dispatch delay (CF read/write `stress` on Linux without IPL modification).

as "trustworthy" OSs. Furthermore, even though the guest OS kernels are isolated, most of the security attacks rely on vulnerability of user level applications. Therefore even if guest OS kernels are isolated, once one of the OSs is attacked, the entire system can be taken over by attacking the other OS via an inter-OS communication channel.

Moreover, relocating OS kernels in privileged mode may degrade the reliability of the system: The kernels running at the same privilege level are able to corrupt each other. However, in a multiOS platform, even though the failure of real-time applications are not propagated to the other part of the system by isolating those OSs, it is a fatal error for the system to continue its service.

These discussions motivated us to remove strong isolation support from our light-weight virtualization layer. To increase the reliability and the security of embedded systems

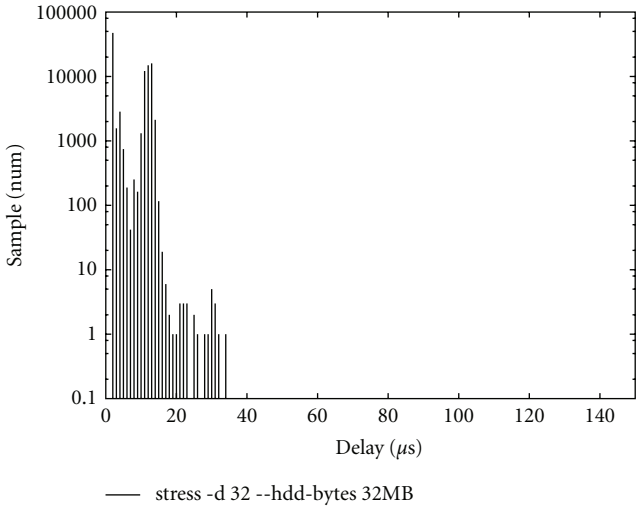

Figure 11: Dispatch delay (CF read/write `stress` on Linux with IPL modification).

with limited computation power and electricity, we need an approach different from desktop and enterprise systems.

6. Conclusion

This paper introduced a method called a composition kernel which constructs an embedded device using both RTOS and GPOS with minimal engineering cost. The approach removes the strong dependencies between OS kernels and instead, let them depend on the underlying thin abstraction layer which requires minimal modification to the OS kernels running on top of it. Our approach executes the abstraction layer and the OS kernels in privileged mode altogether in order to reduce the performance overhead and engineering cost of virtualization. The evaluation shows our approach requires significantly small modifications compared with related work. At the same time it introduces negligible overhead to the real-time responsiveness of the guest RTOS. Furthermore, the method offers flexibility of combining various RTOSs and GPOSs on top of embedded devices with small engineering effort.

References

[1] "FSMLabs: RTLinux," http://www.fsmlabs.com/.

[2] Y. Ishiwata and T. Matsui, "Development of Linux which has advanced real-time processing function," in *Proceedings of the Annual Conference on Robotics Society of Japan (RSJ '98)*, pp. 355–356, 1998.

[3] J. Chen, C. P. Young, D. W. Chang et al., "Building multi-kernel embedded system on pac multi-core platform," in *Proceedings of the International Conference on Quality Software*, pp. 465–472, 2010.

[4] V. Yodaiken, "The RTLinux manifesto," in *Proceedings of the 5th Linux Expo*, 1999.

[5] P. Mantegazza, E. Dozio, and S. Papacharalambous, *RTAI: Real Time Application Interface*, vol. 2000, Specialized Systems Consultants, Seattle, Wash, USA, 2000.

[6] H. Takada, T. Kindaichi, and S. Hachiya, "Linux on ITRON: a hybrid operating system architecture for embedded systems," in *Proceedings of the Symposium on Applications and the Internet (SAINT) Workshops*, IEEE Computer Society, 2002.

[7] M. Humphrey, E. Hilton, and P. Allaire, "Experiences using rt-linux to implement a controller for a high speed magnetic bearing system," in *Proceedings of the 5th IEEE RealTime Technology and Applications Symposium*, pp. 121–130, 1999.

[8] P. Mendoza, J. Vila, I. Ripoll, S. Terrasa, and P. Pérez, "Developing CAN based networks on RT-Linux," in *Proceedings of the 8th International Conference on Emerging Technologies and Factory Automation (ETFA '01)*, pp. 161–167, October 2001.

[9] "Renesas Electronics: SH-4A Software Manual," http://documentation.renesas.com/eng/products/mpumcu/rej09b0003_sh4a.pdf.

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

[11] "Open Kernel Labs: OKL4 community site," http://okl4.org.

[12] I. Molnar, "The realtime preemption patch," 2009, http://www.kernel.org/pub/linux/kernel/projects/rt/.

[13] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole, "A measurement-based analysis of the real-time performance of linux," in *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 133–142, 2002.

[14] R. Goldberg, "Survey of virtual machine research," *IEEE Computer*, vol. 7, no. 6, pp. 34–45, 1974.

[15] P. Barham, B. Dragovic, K. Fraser et al., "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pp. 164–177, ACM Press, 2003.

[16] J. Sugerman, G. Venkitachalam, and B. H. Lim, "Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pp. 1–14, USENIX Association, 2001.

[17] A. Whitaker, M. Shaw, and S. Gribble, "Denali: lightweight virtual machines for distributed and networked applications," in *Proceedings of the USENIX Annual Technical Conference*, pp. 195–209, 2002.

[18] R. Uhlig, G. Neiger, D. Rodgers et al., "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

[19] H. Härtig, M. Hohmuth, J. Liedtke, and S. Schönberg, "The performance of μ-kernelbased systems," in *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 66–77, 1997.

[20] G. Heiser and A. Sydney, "The role of virtualization in embedded systems," in *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems (IIES '08)*, Glasgow, UK, April 2008.

[21] "VirtualLogix: VirtualLogix VLX," http://www.virtuallogix.com/.

[22] K. Yaghmour, "Adaptive domain environment for operating systems," Opersys inc, 2001.

[23] P. Gerum, "Xenomai-Implementing a RTOS emulation framework on GNU/Linux," 2004.

[24] "Amos Waterland: stress," http://weather.ou.edu/~apw/projects/stress/.