Constructing a Multi-OS Platform with Minimal Engineering Cost

Yuki Kinebuchi[1], Takushi Morita[1], Kazuo Makijima[1],
Midori Sugaya[2], Tatsuo Nakajima[1]

[1] Department of Computer Science, Waseda University
{yukikine, morita_t, makijima, tatsuo}@dcl.info.waseda.ac.jp
[2] Dependable Embedded OS Center, Japan Science and Technology Agency (JST)
doly@dependable-os.net

Abstract. Constructing an embedded device with a real-time and a general-purpose operating system has attracted attention as a promising approach to let the device balance real-time responsiveness and rich functionalities. This paper introduces our methodology for constructing such multi-OS platform with minimal engineering cost by assuming asymmetric OS combinations unique to embedded systems. Our methodology consists of two parts. One is a simple hypervisor for multiplexing resources to be shared between operating systems. The other is modifying operating systems to allow them to be aware of each other. We constructed an experimental system executing TOPPERS and Linux simultaneously on a hardware equipped with an SH-4A processor. The modification to each operating system kernel limited to a few dozen lines of code and do not introduce any overhead that would compromise real-time responsiveness or system throughput.

1 Introduction

Software for embedded systems used to be small and simple, but nowadays it is dominating a large part of the system implementation for providing rich functionalities. For instance, modern cell-phones consist of control software (such as a radio transmitter device controller) and rich applications (such as a web browser, a video player, a mailer, etc.). Thus, development with traditional embedded real-time OSes (RTOSes) are unsuited for modern devices. Those traditional RTOSes are only equipped with minimal functionalities, therefore it is hard to meet the strict time-to-market requirements while implementing considerable applications and middleware on top of such RTOSes to provide rich functionalities. This motivated manufacturers to use general-purpose OSes (GPOSes), instead of traditional RTOSes, as the platforms of embedded system software.

However, because GPOSes are unsuited for supporting real-time properties required by embedded systems, some efforts are made to modify them to achieve sufficient real-time responsiveness. The modification to a GPOS kernel requires deep insights into the kernel internal architecture and also requires significant engineering cost. For instance, adding preemption points into a large monolithic kernel requires redesigning various parts of the kernel and could introduce complex timing bugs.

To balance real-time responsiveness and rich functionalities, some approaches constructing an embedded device with both an RTOS and a GPOS have been proposed. One of the approaches is the hybrid system[1, 2]. It is a method to link an RTOS kernel and a functional GPOS kernel together. This approach achieves constructing a system supporting the high real-time responsiveness of an RTOS together with the rich functionalities of a GPOS. However even though this solution is capable of executing real-time applications, the engineering cost of porting existing real-time applications to the real-time layer is problematic for manufacturers. This can be mitigated by using an existing RTOS for a hard-real-time layer like Linux on ITRON[3]. Linux on ITRON has a capability to execute applications developed for the μITRON specification[4].

Typically embedded device manufacturers leverage diverse RTOSes depending on the real-time constraints, the applications set, the software properties they require, etc. Considering various combinations of RTOSes and GPOSes, even though the engineering cost of constructing a single hybrid system is claimed to be small enough, the engineering cost for supporting various combinations of RTOSes and GPOSes would still introduce a great engineering effort.

In this paper, we propose an approach for constructing an embedded device with an RTOS and a GPOS while introducing minimal modifications to both OS kernels by assuming asymmetric OS combinations unique to embedded systems. Our approach is based on a simple hypervisor and paravirtualization like technique. By leveraging the hypervisor, our approach is free to combine various RTOSes and GPOSes. The contributions of this paper are proposing the methodology, and showing its validity by implementing and evaluating it against real-world software and hardware. We developed the hypervisor from scratch, paravirtualized the TOPPERS RTOS and the Linux kernel, and executed them on top of the SH-4A architecture processor. The resulting implementation is simple and efficient enough to accommodate multiple OSes together with a few dozen lines of modifications to both OS kernels while maintaining the real-time responsiveness of the RTOS.

2 Related Work

Various approaches are proposed to balance real-time responsiveness and rich functionalities on a single platform. One of the approaches is modifying a GPOS to support real-time responsiveness. The real-time patch is a modification to a plain Linux kernel to support kernel preemption[5]. It achieves a few hundred μseconds latency[6], but still the result is slower by a factor of ten comparing to typical RTOSes. Even though the mechanism is potentially capable of achieving real-time responsiveness, it could be easily spoiled by a bad-mannered device driver, which holds a lock for a long period. Porting software from an RTOS to Linux would increase the risk of implementing such drivers, because of the difference of programming models between the RTOS and Linux or the developers being unfamiliar with programming on Linux. In addition, porting all the software from the RTOS to Linux would impose substantial engineering cost.

Another approach, known as the hybrid system, is to link an RTOS with a GPOS. RTLinux and RTAI replace the Linux hardware abstraction layer with their own version of RTOSes[1, 2]. Those RTOSes would be executed in privileged mode together with the Linux kernel. The interrupt response time would only be a few μseconds, which is comparable to typical RTOSes. However those microkernels only support their original programming interfaces, which prevents the straight-forward reuse of some real-time software developed for traditional RTOSes. Linux on ITRON is an alternative method to RTLinux and RTAI, which replaces the Linux hardware abstraction layer with an existing RTOS, μITRON[3]. This architecture enables the system to reuse both the software developed for Linux and μITRON. The hybrid system provides high real-time responsiveness comparable with an RTOS with reasonable engineering cost by reusing existing GPOSes. However considering another combination of an RTOS and a GPOS would impose redesigning the hybrid system again from scratch. Because it is usual for manufacturers to leverage diverse OSes, this engineering cost would be problematic.

A virtual machine monitor (VMM) is another technology focusing on accommodating an RTOS and a GPOS into a single embedded device without modifications or with just minimal modifications to the OS kernels[7]. A VMM provides a virtual hardware interface which is identical (or almost identical) to some real hardware and isolation between virtualized guest OSes. To leverage a VMM on embedded systems, developers should consider three trade-offs. First is the trade-off between system throughput and real-time responsiveness, which is a well-known trade-off on system design. Traditional VMMs focus on how to increase the total throughput of workloads provided by guest OSes, because their main targets are enterprise systems or high-performance computing[8, 9]. Thus, they are unsuited for handling real-time properties or supporting embedded system processor architectures. Some VMMs for embedded systems have been developed to meet these real-time requirements on embedded systems. L4 is one of them, and is capable of executing Linux on top of it[10]. Second is the trade-off between full virtualization and paravirtualization. A VMM supporting full virtualization exposes a virtual hardware interface identical to a real hardware interface. OSes can be executed without any modification on full virtualization. However, implementing full virtualization complexifies the design of the VMM itself or requires hardware support for virtualization. Unfortunately hardware support for virtualization is still an unfamiliar feature for embedded system processors. This motivates embedded system VMMs to use paravirtualization for their system design, like L4 did. Third is the trade-off between providing isolation among OSes or not. Strong isolation among guest OSes is an attractive feature for constructing a secure and reliable system. However unlike the VMMs used in the area of enterprise systems, most embedded systems consist of a fixed number of OSes. In addition, as the guest OSes are statically decided by the hardware manufacturer, they can be 'trusted'. This removes the necessity of strong isolation. Without isolation, the design of VMMs would be simpler and their overhead would be smaller.

The previous contributions take a good balance of performance and engineering cost. However their propositions only focus on the combinations of specific RTOSes and GPOSes, and do not consider neither the portability of applications developed for various OSes nor the portability of OSes themselves. From the aspect of accommodating diverse combinations of RTOSes and GPOSes together into a single embedded device, portability should be the primary concern of manufacturers. The advantage of minimizing modifications to OS kernels reduces the possibility of introducing new bugs into virtualized systems. Furthermore, it helps updating the virtualized OSes for bug fixes and security patches.

In order to achieve this requirement while not penalizing performance, our virtualization layer executes guest OS kernels and itself in privileged mode. The virtualization layer multiplexes only minimal hardware resources, while other resources are exclusively assigned to each OS by simply modifying each OS kernel not to access the same devices. Relocating OS kernels in privileged mode degrades the reliability of the system. However, in a multi-OS platform, even though the failure of real-time applications are not propagated to other part of the system, it is a fatal error for the system to continue its service. Recovering from such a real-time application failure with seamless execution is a topic beyond this paper.

3 Design and Implementation

This section introduces our methodology for constructing an embedded device with multiple OSes. The methodology is based on a simple hypervisor called SPUMONE and some modifications to guest OS kernels.

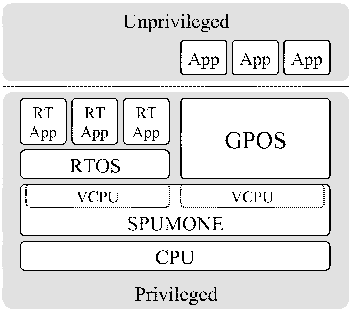

Fig. 1. SPUMONE based system on a single-core processor

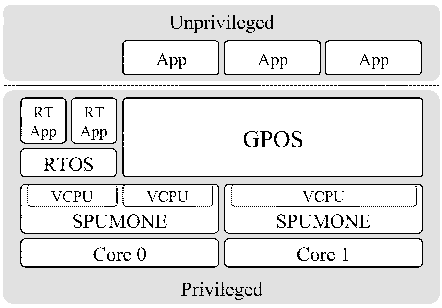

Fig. 2. SPUMONE based system on a multi-core processor

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 traditional hypervisors, SPUMONE itself and guest OSes are executed in privileged mode as shown in Fig.1, in order to simplify the system design and to eliminate the overhead of trapping between privileged and non-privileged mode for system-calls and hypercalls. 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 guest OS kernels. Furthermore it makes the implementation of SPUMONE itself simple. Executing OS kernels in non-privileged mode complicates the implementation of the hypervisor, 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 using a simple function call. Since the interface has no binary compatibility with the original processor interface, we simply modify the source code of guest OS kernels, a method known as the paravirtualization[11, 8]. Thus we assume we have access to the source code of the guest OS kernels. The modifications required to the guest OS kernels are described in details in Sec.3.2.

Virtual Processor Scheduling A processor is multiplexed by scheduling the execution of guest OSes. The execution states of the guest OSes 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 next executing virtual processor. The mechanism is similar to the process paradigm of a classical OS, however the virtual processor saves the entire processor state, including the privileged control registers.

The scheduling algorithm of virtual processors is a fixed priority preemptive scheduling. A virtual processor bound to the RTOS would 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 guest OS 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 if the GPOS is disabling interrupts.

Interrupt/Trap Delivery Interrupt virtualization is a key feature of SPUMONE. Interrupts are investigated by SPUMONE before they are delivered to each OS. SPUMONE receives an interrupt, then looks up the interrupt destination table to see which OS should receive it. The destination virtual processor is statically defined for each interrupt. Traps are also sent to SPUMONE first, then are directly forwarded to the currently executing OS.

To let SPUMONE receive interrupts before the guest OSes, 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. An interrupt is first handled by SPUMONE interrupt handler in which the destination virtual processor is decided and the corresponding scheduler is invoked. 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 the real interrupt occurred, so the code of the guest OS entry points does not need to be modified.

The interrupt enable and disable instructions are also replaced with the virtual instruction interface. A typical OS disables all interrupt sources when disabling interrupts for atomic execution. In our approach, by leveraging the interrupt 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 if it is disabling the interrupts (Fig.3 (1)). On the other hand when the RTOS is running, the interrupts are blocked by the processor (Fig.3 (2)). These blocked interrupts could be sent immediately when the GPOS is dispatched.

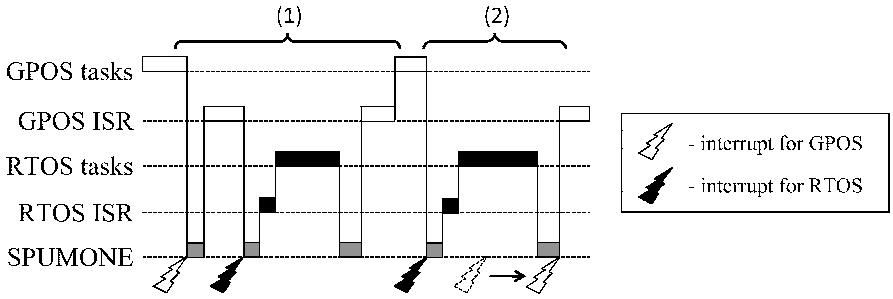

Fig. 3. Interrupt Delivery Mechanism

Multi-core Support SPUMONE also runs on multi-core processors. The design of multi-core SPUMONE is basically the same as the single-core version. As shown in Fig.2, each core is managed by a dedicated SPUMONE instance. Interrupts are handled by the instance bound to each core, then forwarded to the guest OS. Each instance communicates using inter-core interrupt (ICI) and shared memory area. The original processor mechanism of resetting a core is replaced with SPUMONE's function. The development of multi-core SPUMONE is still in progress, so we would not go in details in this paper.

3.2 Modifying OS Kernels

Each OS is modified to be aware of the existence of the other OS, because hardware resources other than the processor are not multiplexed by SPUMONE. Thus those are exclusively assigned to each OS by modifying their kernels. The following describes the points of the OSes to be modified in order to run on top of SPUMONE.

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.

Interrupt Enable and Disable Instruction The instructions enabling and disabling interrupts are typically provided as kernel internal APIs. They 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 priorities 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 4 shows the interrupt priority levels assignment for each OS, which we used in the evaluation environment.

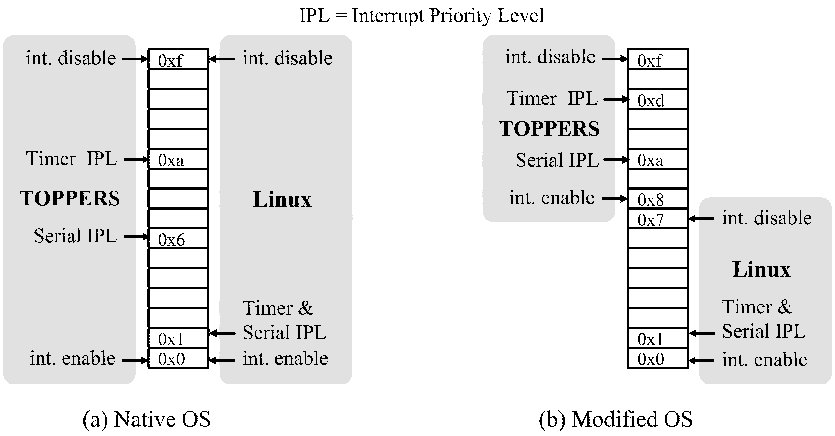

Fig. 4. The interrupt priority levels assignment

Physical Memory A fixed physical memory area is assigned to each OS. The physical address for the guest OSes 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 OSes. From these reasons we assigned fixed physical memory area for each OS.

Idle Instruction On a real processor, the `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.

Peripheral Devices Peripheral devices are assigned by SPUMONE to each OS exclusively. This is done by modifying the configuration of each OS not to share the same peripherals. We assume that most of devices are assigned exclusively to each OS. This assumption is reasonable because embedded system multi-OS platforms have asymmetric OS combinations unlike a symmetric multi-OS platform for enterprise systems. It consists of different kinds of OSes, usually an RTOS and a GPOS. 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, only one interrupt controller is provided by the experimental processor we used. Usually the OS clears some of its 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 basic overhead, the engineering cost of modifying the guest OS kernels, and 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 128MB memory. We use TOPPERS/JSP 1.3 as RTOS and Linux 2.6.20.1 as GPOS. Linux mounts an NFS share exported by the host machine as its root file system.

4.1 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 1 shows the average and the worst case CPU cycles required to handle the interrupts sent to native TOPPERS and modified TOPPERS. In the average case SPUMONE imposes $0.67\mu s$ 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 1. 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 on SPUMONE	average	367	0.92	0.67
	worst	1582	3.96	3.70

Table 2 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 each 1ms, and executes no other tasks. The result shows that SPUMONE and TOPPERS impose 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.

Table 2. Linux kernel build time

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

4.2 Engineering Cost

We evaluated 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 3 is a list of the modified files in Linux. Table 4 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 because we are executing guest OS in privileged mode.

Table 3. 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 (64MB) 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

Table 4. The total number of modified LOC in *.c, *.S, *.h, Makefiles

OS	Added LOC	Removed LOC
Linux on SPUMONE (Linux 2.6.20.1)	56	17
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	-

4.3 Effect of Linux Load to TOPPERS Real-time Properties

We measured the effect of Linux load to TOPPERS periodic task execution intervals. Only the periodic task is executed on TOPPERS. Figure 5, 6, 7, 8 shows the frequency distribution of the intervals of the 1ms periodic task running on TOPPERS. Figure 7 and 8 are measured with running the `stress` command on Linux to show the effect of the CPU load and the I/O load. CPU load repeat invoking `sqrt()`. I/O load repeats invoking `sync()`, which triggers flushing data to the file system. The intervals are sampled 100,000 times each. Note that the y-axis is showed in log scale. The overhead of switching from Linux to TOPPERS and execution inside SPUMONE would delay the start-up of the periodic task, which could be the cause of jitters. The maximum error for delay was $20\mu s$ showed in Fig.8. The results show the jitters are small, however we need further investigations to explain the cause of the jitters.

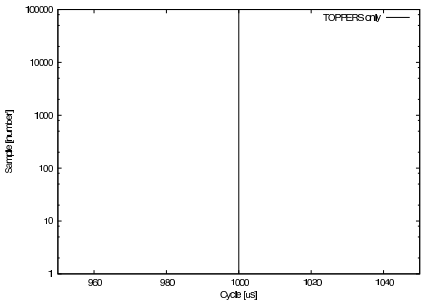

Fig. 5. The frequency distribution of the periodic task execution intervals. TOPPERS only. Only the periodic task running.

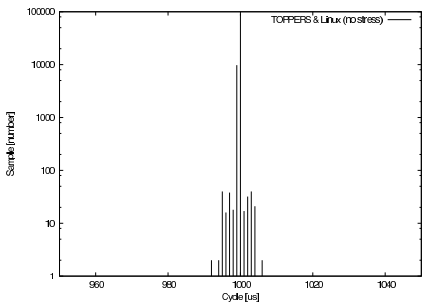

Fig. 6. The frequency distribution of the periodic task execution intervals. TOPPERS and Linux with no load on SPUMONE.

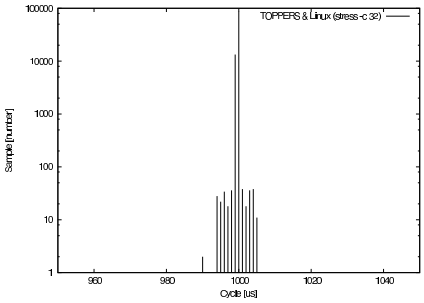

Fig. 7. The frequency distribution of the periodic task execution intervals. TOPPERS and Linux with CPU load on SPUMONE.

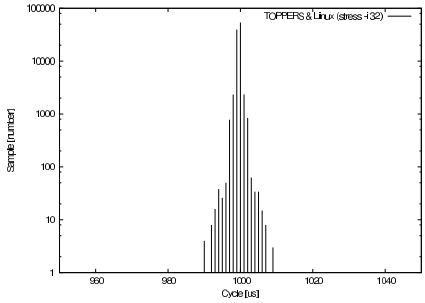

Fig. 8. The frequency distribution of the periodic task execution intervals. TOPPERS and Linux with I/O load on SPUMONE.

5 Conclusion

One of the primary requirements for constructing a hybrid system for embedded system is engineering cost. Existing research only focused on the engineering cost of a specific combination of RTOSes and GPOSes, however those approaches did not consider diverse combinations of OSes. This paper introduced our approach to construct an embedded device with an RTOS and a GPOS with minimal engineering cost, which can be adapted to various OS kernels in the similar way. The approach is based on utilizing the thin SPUMONE virtualization layer and modifying a few parts of the guest OS kernels, a method known as paravirtualization. Our approach executes the virtualization layer and the guest OS kernels in privileged mode altogether in order to reduce the performance over head engineering cost of virtualization. The evaluation shows our approach requires significantly small modifications with introducing reasonable overhead to the real-time responsiveness of the guest RTOS, which allows the freedom of combining various RTOSes and GPOSes to run on top of embedded devices.

References

1. Yodaiken, V.: The RTLinux Manifesto. In: Proc. of The 5th Linux Expo. (1999)
2. Mantegazza, P., Dozio, E., Papacharalambous, S.: RTAI: Real Time Application Interface. Volume 2000., Specialized Systems Consultants, Inc. Seattle, WA, USA (2000)
3. Takada, H., Kindaichi, T., Hachiya, S.: Linux on ITRON: A Hybrid Operating System Architecture for Embedded Systems. In: Proceedings of the 2002 Symposium on Applications and the Internet (SAINT) Workshops, IEEE Computer Society Washington, DC, USA (2002)
4. ITRON Project: μitron4.0 specification. `http://www.ertl.jp/ITRON/`
5. Molnar, I.: The realtime preemption patch. `http://www.kernel.org/pub/linux/kernel/projects/rt/` (2009)
6. Abeni, L., Goel, A., Krasic, C., Snow, J., Walpole, J.: A measurement-based analysis of the real-time performance of linux. Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE (2002) 133–142
7. Heiser, G., Sydney, A.: The role of virtualization in embedded systems. 1st IIES, Glasgow, UK, Apr (2008)
8. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, New York, NY, USA, ACM Press (2003) 164–177
9. Sugerman, J., Venkitachalam, G., Lim, B.H.: Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In: Proceedings of the General Track: 2002 USENIX Annual Technical Conference, Berkeley, CA, USA, USENIX Association (2001) 1–14
10. Leslie, B., van Schaik, C., Heiser, G.: Wombat: A portable user-mode Linux for embedded systems. Proceedings of the 6th Linux. Conf. Au (2005)
11. Whitaker, A., Shaw, M., Gribble, S.: Denali: Lightweight virtual machines for distributed and networked applications. Proceedings of the USENIX Annual Technical Conference (2002) 195–209