A STUDY ON REAL-TIME RESPONSIVENESS ON VIRTUALIZATION BASED MULTI-OS EMBEDDED SYSTEMS

Yuki Kinebuchi, Hitoshi Mitake, Yohei Yasukawa, Takushi Morita
Alexandre Courbot and Tatsuo Nakajima
Department of Computer Science and Engineering, Waseda University, Tokyo, Japan

Keywords: Embedded and Real-Time Systems, Multi-core processors, Virtualization Techniques.

Abstract: Despite the strong requirement of supporting deterministic real-time scheduling on virtualization based multi-OS embedded systems, which enables co-location of a real-time operating system and a general-purpose operating system on a single device, there are few investigations in the real-world hardware. In this paper we introduce our virtualization layer called SPUMONE, which runs on single-core and multi-core SH-4A processors. SPUMONE achieves the low overhead, and requires a small amount of engineering efforts to modify guest OS kernels for executing on SPUMONE. SPUMONE now can execute the TOPPERS real-time OS and Linux as a general-purpose OS concurrently on a single embedded platform. In addition we propose two techniques to mitigate the interference of Linux to the real-time responsiveness of RTOS. The first technique leverages the interrupt priority level mechanism supported by the SH-4A processor. The second is the proactive migration of a virtual core among physical cores to prevent the Linux kernel activity from blocking the interrupts assigned to RTOS. The evaluation shows that our techniques can decrease the interrupt latency of RTOS entailed by Linux. In addition, sharing a physical core between RTOS and Linux will increase total processor utilization.

1 INTRODUCTION

Modern embedded systems like cell-phones and digital home appliances are rapidly enhancing their functionality, getting competitive with desktop systems. However there are some embedded system specific requirements for real-time control processing, which is difficult to be supported by general-purpose operating systems.

Therefore, constructing an embedded device with a real-time operating system (RTOS) and a general-purpose operating system (GPOS) has been attracted as an approach to let embedded devices balance real-time responsiveness and rich functionalities. There are various approaches to satisfy the above requirement. One of the approaches is to use a multi-core SoC typically equipped with two independent processors, one for RTOS and the other for GPOS. Another approach is to use the hybrid system (Mantegazza, 2000); (Takada, 2002); (Yodaiken, 1999) which executes GPOS as a task of RTOS.

In this paper, we focus on virtualization technologies, originally widely used in enterprise servers and desktop computers. Now, embedded systems are attractive target as a new research field of virtualization technologies (Heiser, 2008). Embedded systems require different characteristics and gives some new challenges that have not been discussed in the previous research fields of virtualization technologies. According to the discussions in (Armand, 2009), the requirements to the embedded system hardware virtualization are:

i. To require minimal or no modification to OS kernels and their applications.
ii. To let OSes to reuse their native device drivers.
iii. To support the real-time responsiveness in order to maintain the real-time property of RTOS.

Virtualization technologies for enterprise servers and desktop systems, like VMware (http://www.vmware.com/) and Xen (Barham, 2003), do not

Kinebuchi Y., Mitake H., Yasukawa Y., Morita T., Courbot A. and Nakajima T..
A STUDY ON REAL-TIME RESPONSIVENESS ON VIRTUALIZATION BASED MULTI-OS EMBEDDED SYSTEMS .
DOI: 10.5220/0003319903690378
In *Proceedings of the 1st International Conference on Pervasive and Embedded Computing and Communication Systems* (PECCS-2011), pages 369-378
ISBN: 978-989-8425-48-5
Copyright © 2011 SCITEPRESS (Science and Technology Publications, Lda.)

fulfill these requirements. Especially the third requirement is difficult to be supported by traditional virtualization technologies. Because the virtual memory virtualization and the I/O virtualization require complex manipulation of data structures inside virtualization layers, they require to synchronize the data structures, and make the virtualization layer complex. Therefore, we need to develop virtualization layers specialized for embedded systems. In our approach, we have developed our own virtualization layer for embedded devices and evaluated its real-time responsiveness.

There are three contributions introduced in this paper.

- The first contribution is an OS consolidation methodology which fits the requirements of embedded systems. The evaluation shows that the basic overhead and engineering cost required to the guest OSes are significantly smaller compared with other solutions.
- The second contribution is an investigation on the real-time properties of the virtualization technology for embedded devices. Despite the growth of real-time virtualization technologies, their real-time properties have not been sufficiently discussed.
- The last contribution is to propose two techniques for decreasing the latency introduced to RTOS. The first technique is to leverage the interrupt priority level (IPL) mechanism to enable RTOS to preempt a GPOS's critical section at any time. The other is to migrate virtual cores among physical cores, when they enter into a critical section, in order to prevent GPOS kernel activities to block the execution of RTOS.

We have developed a thin virtualization layer called SPUMONE which enables the co-execution of multiple OSes on a single-core processor and multi-core processor equipped with the SH-4A architecture cores. SPUMONE can co-execute TOPPERS RTOS (TOPPERS is a RTOS which meets μITRON RTOS specification widely used in Japanese industry) and Linux. The evaluation shows that our approach improves the real-time responsiveness significantly. Also, our evaluation shows that the tradeoffs between the proposed techniques that offer a guideline to consider which technique is appropriate for respective target domains.

The rest of the paper is structured as follows. In Section 2, we show the design and implementation of SPUMONE. Section 3 proposes two techniques to reduce interrupt latency. Section 4 presents the evaluation showing the effectiveness of the proposed approach. In Section 5, we show related work, and finally, we conclude the paper in Section 6.

2 DESIGN AND IMPLEMENTATION

This section introduces our methodology for accommodating multiple OSes on the top of a single embedded device. The methodology is based on a thin virtualization layer called SPUMONE and some modifications to guest OS kernels.

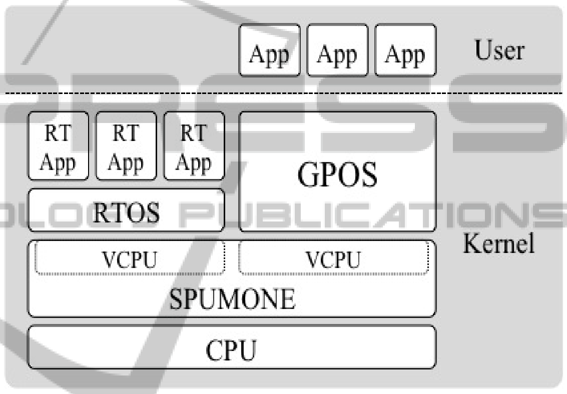

Figure 1: SPUMONE based system on a single-core processor.

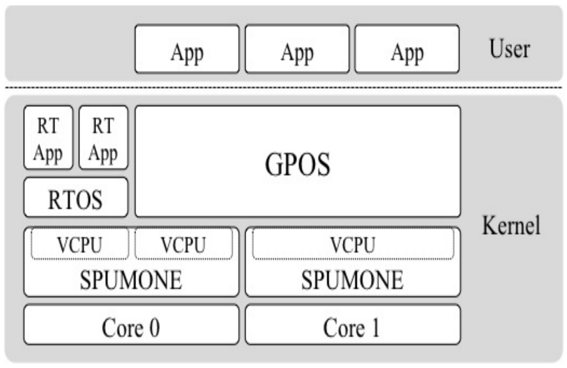

Figure 2: SPUMONE based system on a multi-core processor.

2.1 Light-weighted Virtualization Layer: for Embedded Systems: 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 processors. In other words, SPUMONE provides a virtual multi-core processor interface on the top of a physical single-core processor. Unlike typical virtualization layers, SPUMONE itself and guest OS kernels are executed in the privileged address space as shown in Fig. 1, in order to simplify the system design and to eliminate the overhead of cross domain calls between the user and kernel mode for invoking system-calls and hypercalls. If an OS does not leverage privilege levels, its applications will be executed in the kernel mode altogether. Executing OS kernels in the user mode is known to complicate the implementation of the virtualization layer, because all privileged instructions need to be emulated.

In SPUMONE, the majority of the kernel instructions, including the privileged instructions, are executed directly by the real processor, and only a minimal set of instructions are emulated by SPUMONE. The emulated instructions are invoked from the OS kernels using simple function calls. 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 the para-virtualization. Thus we assume that we have the access to the source code of the OS kernels. However, the modifications of OS kernels are very small as described in Section 2.2.

SPUMONE assumes to use an SMP (Symmetric Multiprocessing)-based multi-core processor. All codes and data for applications and guest OS kernels reside in the shared memory. SPUMONE for multi-core processors is designed in a distributed model: a dedicated instance of SPUMONE is assigned to each physical core as shown in Fig. 2. This design is chosen in order to eliminate the unpredictable overhead of synchronization among multiple processor cores. In addition, the basic lock mechanism can be easily shared between the single-core and multi-core version. The approach simplifies the design of SPUMONE. It also enables the system to scale on multi-core and many-core processors as discussed in (Baumann, 2009).

2.1.1 Interrupt/Trap Delivery

Interrupt virtualization is a key feature of SPUMONE. Interrupts are intercepted by SPUMONE before they are delivered to each guest OS. When SPUMONE receives an interrupt, it looks up the interrupt destination table to make a decision to which OS it should be delivered. The destination virtual processor is statically defined for each interrupt source when the OS kernels are built. Traps are also delivered to SPUMONE first, then are directly forwarded to the currently executing OS.

To let SPUMONE receive interrupts before the OSes, we modified the interrupt entry point of the OS kernels to the 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 examined by the SPUMONE's interrupt handler in which the destination virtual processor is decided and the corresponding scheduler is invoked. When the interrupt triggers OS switching, all the registers of the current OS are saved into the register stack, then the register stack for the previous OS is restored. Finally, the execution branches into the entry point of the destination OS. The processor registers are setup just as the real interrupt is occurred, so the source code of the OS entry points does not need to be modified.

The interrupt delivery process on a multi-core platform works basically as same as the one on a single-core platform. Each SPUMONE instance delivers interrupts to their destinations. On a multi-core system, virtual cores may migrate among physical cores. In order to deliver interrupts to a virtual core running on a different core, the assignments of interrupts and physical cores are switched along with the virtual core migration.

2.1.2 Virtual Processor Scheduling

A processor is multiplexed by scheduling the execution of OSes. The execution states of the OSes are managed by data structures that we call virtual processors or virtual cores. When switching the execution of the virtual processors, all the hardware registers are stored into the corresponding virtual processor's register table, and then restored from the table of the next executing virtual processor. The mechanism is similar to the process implementation of a typical OS, however the virtual processor saves the entire processor state, including the privileged control registers.

The scheduling algorithm of virtual processors is the fixed priority preemptive scheduling. When RTOS and GPOS share the same physical core, the virtual processor bound to RTOS would gain a higher priority than the virtual processor bound to GPOS in order to maintain the real-time responsiveness of RTOS. This means that GPOS is executed only when the virtual processor for RTOS is in an idle state and has no task to be executed. The process scheduling is left up to OSes so the scheduling model for each OS is not changed. Idle RTOS resumes its execution when it receives an interrupt. The interrupt for RTOS preempts GPOS immediately, even if GPOS is disabling its interrupts.

When virtual cores assigned to GPOS are migrated to be executed on a shared core, those cores are scheduled with the timesharing scheduler.

2.1.3 Inter-core Communication

Communications among SPUMONE instances running on their physical cores are implemented with the shared memory area and the inter-core interrupt (ICI) mechanism. First, a sender stores data on a specific memory area, then it sends an interrupt to a receiver, and the receiver copies the data from the shared memory.

2.2 Modifying OS Kernels

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

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

Bootstrap.
In addition to the features supported by the single-core SPUMONE, the multi-core version provides the virtual reset vector device, which is responsible for resetting the program counter of the virtual core that resides on a different core.

Physical Memory.
A fixed physical memory area is assigned to each guest OS. The physical address for the OSes can be simply changed by modifying the configuration files or their source codes. Virtualizing the physical memory would increase the size of the virtualization layer and the substantial performance overhead. In addition, unlike the virtualization layer for enterprise systems, embedded systems need to support a fixed number of OSes. For these reasons we assigned the fixed physical memory area for each OS.

Idle Instruction.
On a real processor, the idle instruction suspends a processor until it receives an interrupt. On a virtualized environment, this is used to yield the use of real processor to another 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 the 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 can be 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 RTOS and GPOS. For instance, RTOS is used for controlling specific peripherals such as a radio transmitter and some digital signal processors, and GPOS for controlling a display and various human interaction devices.

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 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.

3 INTERRUPT LATENCY REDUCTION

3.1 Interrupt Priority Level Separation

In order to minimize the interrupt latency of RTOS in the reasonable bound although the activities of GPOS run concurrently on a single device, we propose two technique. The first technique is replacing the interrupt enable and disable instructions with the virtual instruction interface. A typical OS disables all interrupt sources when disabling interrupts for the atomic execution. On the other hand, our approach leverages the interrupt mechanism of the processor: we assign the higher half of the interrupt priority levels (IPLs) to RTOS and the lower half to GPOS (Fig. 3). When GPOS tries to block the interrupts, it modifies its interrupt mask to the middle priority. RTOS may therefore preempt GPOS even if it is disabling the interrupts

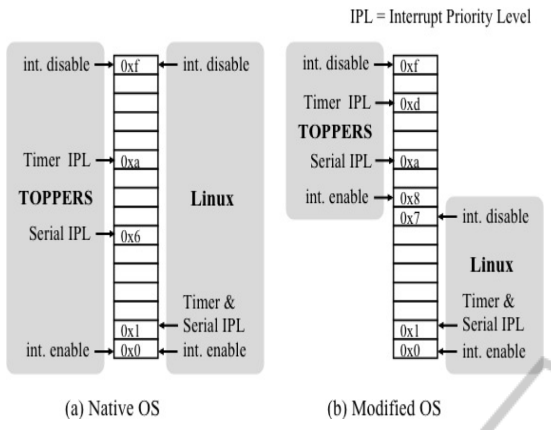

Figure 3: The interrupt priority levels separation.

(Fig. 4 (1)). On the other hand, when RTOS is running, the interrupts are blocked by the processor (Fig. 4 (2)). These blocked interrupts could be delivered immediately when GPOS is dispatched.

The instructions enabling and disabling interrupts are typically provided as the kernel internal API. They are typically coded as inline functions or macros in the kernel source code. For GPOS, we replace those APIs with the instructions enabling the entire level of interrupts and disabling only low priorities interrupts. For RTOS, we replace the API with the instructions enabling only high priority interrupts and disabling the entire level of interrupts. Therefore, interrupts assigned to RTOS are immediately delivered to RTOS, and the interrupts assigned to GPOS are blocked during the RTOS's execution. Fig. 3 shows the interrupt priority levels assignment for each OS, which we used in the evaluation environment.

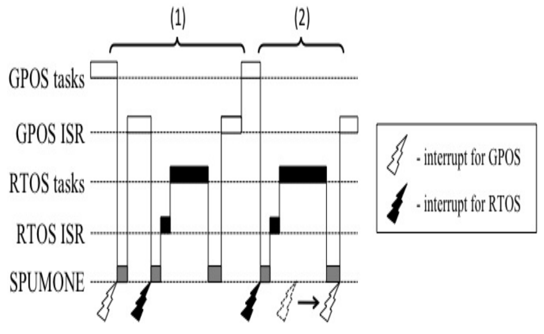

Figure 4: Interrupt Delivery Mechanism.

3.2 Virtual Processor Core Migration

The second technique is based on the virtual core migration. When we implemented the first technique, we found that some paths of the GPOS kernel gained a highest lock priority unexpectedly (e.g. bootstrap, idle thread). This suggests us the possibility that some device drivers or kernel modules programmed in a bad manner gains a higher IPL and interferes with the activity of RTOS. We modified SPUMONE to proactively migrate a virtual core, which is assigned to GPOS sharing a physical core with RTOS, to another physical core when it traps into the kernel or interrupts are triggered as shown in Fig. 5. In this way, only the user level code of GPOS is executed concurrently on a shared physical core, which will never change the priority levels. Therefore, RTOS may preempt GPOS immediately without separating IPLs used in the first technique.

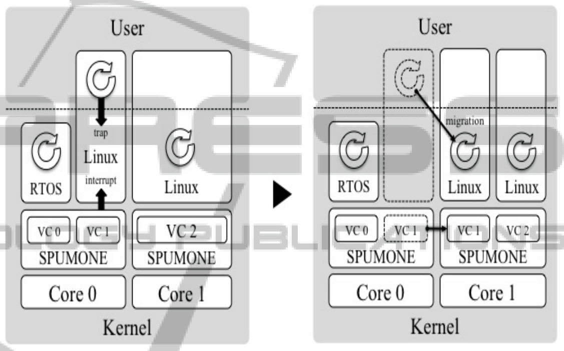

Figure 5: Virtual core migration.

4 EVALUATION

We evaluated the basic overhead, the engineering cost of modifying the OS kernels, and the real-time responsiveness of RTOS running on SPUMONE. The evaluation for a single-core system is done on the SH-2007 reference board, with the SH-4A 400 MHz processor and 128MB memory. The evaluation for a multi-core system is done on the MSRP1BASE02, with a RP1 quad core 600 MHz processor and 128MB memory. The core is also based on the SH4A architecture. We use TOPPERS/JSP 1.3 as RTOS and Linux 2.6.16 as GPOS for the single-core, and Linux 2.6.24.3 as GPOS for the multi-core processor. Linux mounts an NFS share exported by the host machine as its root file system. The basic overhead and engineering cost are both evaluated on single-core environment.

4.1 Basic Overhead

For evaluating the basic overhead of SPUMONE, we have measured the overhead of interrupt handling latency and the time to build the Linux kernel on the top of native (an unmodified OS running on a bare-metal hardware) Linux and modified Linux, respectively.

Table 1 shows the average and the worst case CPU cycles required to handle the interrupts delivered to native TOPPERS and modified TOPPERS. In the average case, SPUMONE imposes 0.67*μs* overhead to the latency. The worst case overhead shows the time required to save the states of Linux and to restore the states of TOPPERS. The increased latency is sufficiently small and predictable for executing real-time applications.

Table 1: The latency of handling the timer interrupts in TOPPERS.

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

Table 2 shows the time required to build Linux kernel on native Linux and modified Linux executed on the top of SPUMONE together with TOPPERS. TOPPERS only receives the timer interrupts each 1*ms*, and executes no other tasks. The result shows that SPUMONE and TOPPERS impose the overhead of 1.4% to the 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	68m5.9s	-
Linux and TOPPERS	69m3.1s	1.4%

4.2 Engineering Cost

We evaluated the engineering cost of reusing RTOS and GPOS by comparing the number of modified lines of code (LoC) in each OS kernel. Table 3 shows the LoC added and removed from the original Linux kernels. We did not count the lines of device drivers for inter-kernel communication because the number of lines will differ depending on how many protocols they support and how complex are them. We did not include the LoC of utility device drivers provided for communication between Linux and RTOS or Linux and servers processes because it depends on how many protocols and how complex those are implemented.

Table 3: The total number of modified LoC in *.c, *.S, *.h, Makefiles.

OS(Linux version)	Added LoC	Removed Loc
Linux/SPUMONE(2.6.24.3)	161	8
RTLinux 3.2(2.6.9)	2798	1131
RTAI 3.6.2 (2.6.19)	5920	163
OK Linux (2.6.24)	28149	-

The table also shows the modified LoC for RTLinux (Yodaiken 1999), RTAI (Mantegazza 2000) and OK Linux (Heiser 2008) that are previous approaches to support the multi OS environments. Since we could not find RTLinux, RTAI, OK Linux for the SH-4A processor architecture, we evaluated them developed for the Intel 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 OS in the kernel mode.

4.3 The Effect of Linux Load to TOPPERS Real-time Responsiveness

We measured the effect of loads on Linux to the dispatch latency of a periodic task in TOPPERS. We compared two proposed techniques to reduce the interrupt response time.

A periodic task runs every 1*ms*. It is sampled 100,000 times during the measurement. The dispatch latency is the time spent from the interrupt triggered until the periodic task starts its execution. Only the periodic task is executed on TOPPERS which means that no other task on TOPPERS will prevent the execution of the periodic task.

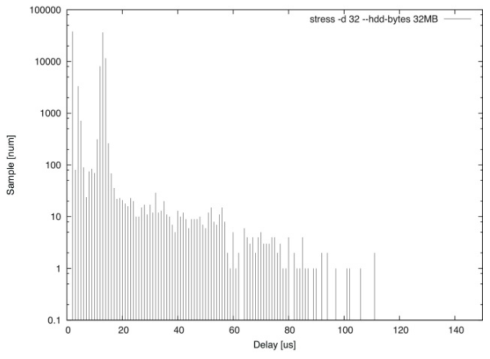

Figure 6: Dispatch latency on single core (CF write stress on Linux without the IPL separation technique).

Fig. 6 and 7 compares the distribution of the timer interrupt latency without and with the IPL separation technique under invoking continuous write () to a CF card file system. We executed a stress program as the workload on the top of Linux. The measurement with the file system load shows the maximum latency of 111μs without the IPL separation technique. With the IPL separation technique, this latency is decreased to 34μs. Comparing this result with the measurement done by (Abeni, 2002), with the 1.8GHz Athlon processor which shows the maximum latency of a few hundred μs, we can see that our measurement with the 400MHz SH processor achieves fairly small dispatch latency.

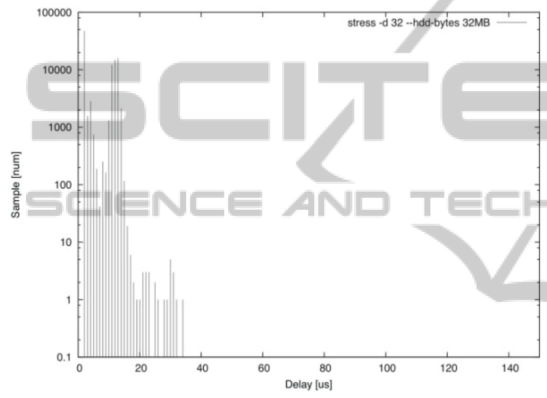

Figure 7: Dispatch latency on single core (CF write stress on Linux with the IPL separation technique).

Fig. 8 and 9 compares the distribution of the timer interrupt latency without and with the virtual core migration technique under invoking continuous write () to NFS share file system. The measurement without the virtual core migration technique shows the maximum latency of 96 μs. With the virtual core migration technique is enabled, the maximum latency is reduced to 39 μs.

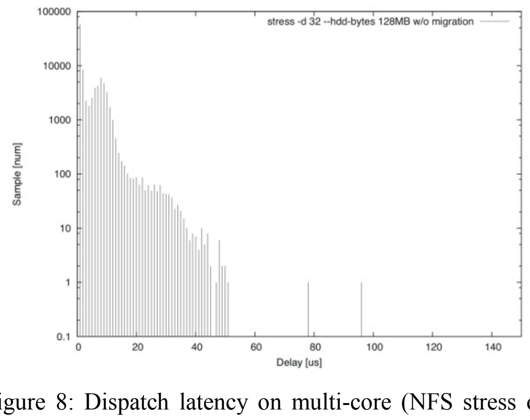

Figure 8: Dispatch latency on multi-core (NFS stress on Linux without the virtual core migration technique).

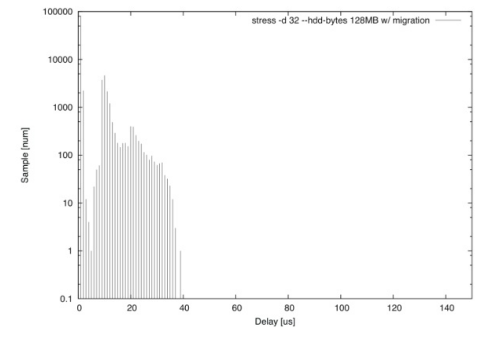

Figure 9: Dispatch latency on multi-core (NFS stress on Linux with the virtual core migration technique).

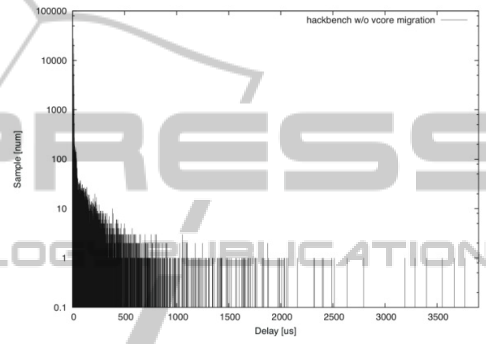

Figure 10: Dispatch latency on multi-core (frequent IPC on Linux without the virtual core migration technique).

Fig. 10 and 11 compares the distribution of the dispatch latency without and with the virtual core migration technique under the frequent IPC load on the top of Linux. The IPC load is generated by hackbench, which is modified to acquire clock cycles from a device file which returns the correct count independent of the processor utilization of RTOS. The latency measured without the virtual core migration technique numbered 3770 μs. This is because the interrupt assigned to RTOS is blocked by the spinlock mechanism of Linux. When the virtual core migration technique is enabled, the interrupt latency is reduced to 44 μs.

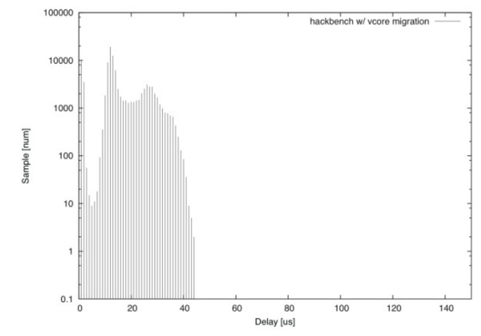

Figure 11: Dispatch latency on multicore (frequent IPC on Linux with the virtual core migration technique).

The overall measurement shows the IPL separation technique and the virtual core migration technique mitigates the effect of interrupt disabling performed inside the Linux kernel. Even though this measurement only shows the statistical maximum interrupt latency, it is clear that the proposed techniques can reduce the average interrupt latency significantly.

4.4 The Effect of TOPPERS Periodic Task Load to Linux throughput

We have also measured the effect of the processor utilization of TOPPERS to Linux. We compared the score of the Dhrystone benchmark and the hackbench benchmark with Linux running on the top of 4 dedicated cores (indicated as *4 cores* in the Fig 12 and 13), Linux running on the top of 3 dedicated cores and one core shared with TOPPERS in various workloads (*xx%* in the figures), and Linux running on the top of 3 dedicated cores (indicated as *3 cores* in the figures). The task on TOPPERS is executed in the cycle of 10 *ms*. The percentage shows the ratio of the execution time of the periodic task against the cycle (*30%* means that the task is executed for 3 *ms* continuously).

Fig. 12 shows the total score of the Dhrystone benchmark. The bar at the left end shows the score of the evaluation done with Linux executed on the top of SPUMONE with three physical cores. As long as the workload of the periodic task grows, the score of Dhrystone degrades. At the load of 90%, the result gets close or less than the score of the three dedicated core configuration. The result shows the overhead of the virtual core migration technique is not significant in the benchmark.

In contrast, Fig. 13 shows the score of hackbench, the benchmark which evaluates the scalability of the number of cores. The execution time of hackbench is increased when the virtual core migration technique is enabled. This is caused by the frequent system calls invoked during the benchmark, which triggers a virtual core to migrate among physical cores very frequently.

From the point of the processor utilization, it is better to let Linux share a physical core with RTOS. Since RTOS application processes are usually designed not to consume the entire processor time, in many cases, there is free processor time that can be used by Linux effectively. However, the result of hackbench shows that the performance improvement depends on the characteristics of a workload running on the top of Linux.

The results show that we need to assign Linux processes carefully to virtual cores when some Linux processes invoke system calls very frequently. In this case, SPUMONE should not execute the virtual core with RTOS. Because the manual configuration between virtual cores and physical cores by considering the number of system call invocations, it is possible to enhance SPUMONE to implement the above manual policy without any programmers' efforts. For example, when SPUMONE finds the number of invoking Linux system calls is increased, the virtual core to execute the system calls is migrated to another physical core that is not shared with RTOS, and the virtual core to invoke less system calls is migrated to the physical core that is shared with RTOS.

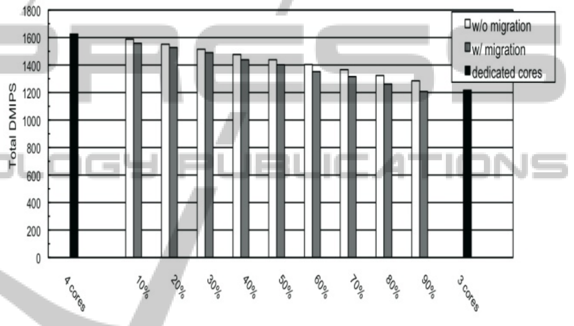

Figure 12: The effect of load on TOPPERS to Linux's DMIPS score (y-axis in DMIPS, larger is better).

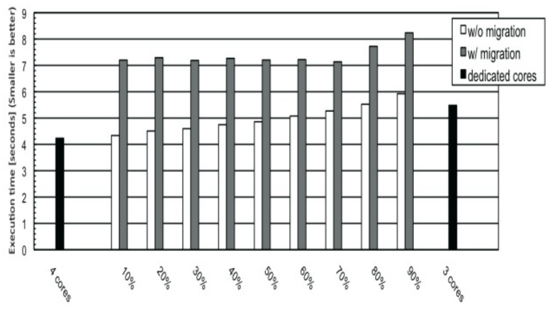

Figure 13: The effect of load on TOPPERS to Linux's hackbench (y-axis in seconds, smaller is better).

5 RELATED WORK

Various approaches have been proposed to balance real-time responsiveness and rich functionalities on a single platform. One of the approaches is modifying GPOS to support real-time responsiveness. The real-time patch is a modification to the plain Linux kernel to support the kernel preemption (Molnar, 2010). It achieves a few hundred *μs* latency (Abeni, 2002), but still the result is not enough 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 bad-mannered device drivers, which disable interrupts for a long period. Porting existing programs from RTOS to Linux would increase the risk of implementing such device drivers, due to the differences between the programming models of RTOS and Linux. Also, the developers usually using RTOS are unfamiliar with programming on Linux. Then, the possibility to write bad mannered device drivers becomes high. In addition, porting all the software from RTOS to Linux would impose the substantial engineering cost.

Another approach, known as the hybrid system, is to execute RTOS in the GPOS kernel. RTLinux and RTAI replace the Linux hardware abstraction layer with their own version of RTOSes (Mantegazza, 2000); (Yodaiken, 1999). Those RTOSes would be executed in the kernel mode together with the Linux kernel. The interrupt response time would only be a few *μs*, which is comparable to typical RTOSes. However those RTOSes only support their original programming interfaces, which prevents the straight-forward reuse of some existing real-time software developed for traditional RTOSes. Linux on ITRON is an alternative system to RTLinux and RTAI, which replaces the Linux hardware abstraction layer with the existing RTOS, *μ*ITRON (Takada, 2002). This architecture enables the system to reuse both the software developed for Linux and *μ*ITRON. The hybrid system provides the high real-time responsiveness comparable with RTOS with the reasonable engineering cost because a large amount of existing software for embedded systems in Japan has been developed on *μ*ITRON. However, considering another combination of RTOS and GPOS would impose redesigning the hybrid system again from scratch. Because it is usual for manufacturers to leverage diverse RTOSes, this engineering cost would be problematic.

A virtual machine monitor (VMM) is another technology focusing on accommodating RTOS and GPOS into a single embedded device without the modifications or with just the minimal modifications to the OS kernels (Heiser, 2008). VMM provides a virtual hardware interface which is identical (or almost identical) to some real hardware and the isolation mechanism between virtualized guest OSes. VMM supporting the full-virtualization technique exposes a virtual hardware interface identical to a real hardware interface. OSes can be executed without any modification on the full-virtualization based VMM. However, implementing the full-virtualization technique complicates the design of VMM itself or requires special hardware supports for the hardware virtualization. Unfortunately, the hardware supports for the hardware virtualization is still an unfamiliar feature for embedded system processors. This motivates VMM for embedded system to use the para-virtualization technique. The L4 microkernel is a typical system to offer the para-virtualization interface for embedded system. However, the engineering cost required for para-virtualizing a guest OS kernel is also problematic as described in Section 4.2. In addition, switching the privilege levels between a guest OS and VMM will entail the significant performance degradation.

In order to achieve the low engineering cost while not penalizing performance, SPUMONE executes OS kernels and itself in the privileged mode. This also contributes to reduce the engineering cost of modifying OS kernels, because the majority of privileged instructions can be executed by a processor directly and only a minimal set of instructions needs to be emulated. Furthermore, SPUMONE 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 hardware resources.

There are some researches on how to design scalable OSes on multi-core and many-core processors. Corey is a many-core operating which allows applications to explicitly specify the assignment of critical OS kernel data structures among cores (Wickizer, 2008). This hints the kernel to schedule processes to improve the cache locality. Multikernel is an experimental OS kernel which exploits the multi-core and many-core processor parallelism by constructing the system with the distributed model (Baumann, 2009). SPUMONE's basic design is similar to Multikernel, but our contribution is to reuse existing software programs developed on the top of various existing OSes while satisfying the real-time responsiveness.

6 CONCLUSIONS

In this paper we proposed a light-weight virtualization layer which achieves the low overhead and low engineering cost to construct multi-OS embedded systems. In addition, we evaluated the real-time responsiveness of RTOS running concurrently with Linux under various workloads. We proposed two techniques to mitigate the performance interference from Linux to RTOS; the IPL separation technique and the virtual core migration technique. The evaluation shows that our techniques reduced the interrupt latency significantly. Especially on the multi-core system, Linux sharing a physical core with RTOS increases the processor utilization. However with an application triggering frequent system calls may loses its throughput due to the frequent virtual core migration among physical cores.

In the future, we will implement several dynamic policies to map virtual cores and physical cores according to the system workloads. The policies migrate virtual cores according to the number of system call invocations of Linux as described in Section 4.2. The dynamic mapping policy between virtual and physical cores also offers the possibility to reduce power consumption significantly by migrating all virtual cores to a small number of physical cores while the system workload is very low, because it is possible to turn off the power of most of physical cores.

REFERENCES

L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole. A measurement-based analysis of the real-time performance of linux. *Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE*, pages 133–142, 2002.

F. Armand and M. Gien. A practical look at microkernels and virtual machine monitors. In *Proceedings of the 6th Consumer Communications and Networking Conference(IEEE CCNC'09), Las Vegas, NV, USA*, 2009.

P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A.Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. SchÅNupbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

G. Heiser and A. Sydney. The role of virtualization in embedded systems. *1st IIES, Glasgow, UK, Apr*, 2008.

P. Mantegazza, E. Dozio, and S. Papacharalambous. RTAI: Real Time Application Interface. In *Linux Journal*, volume 2000. Specialized Systems Consultants, Inc. Seattle, WA, USA, 2000.

I. Molnar. The realtime preemption patch. http://people.redhat.com/mingo/realtime-preempt/.

H. Takada, T. Kindaichi, and S. Hachiya. 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.

Silas B. Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang and Zheng Zhang, "Corey: An Operating System for Many Cores", *USENIX OSDI 2008 (Operating Systems), 2008.*

V. Yodaiken. The RTLinux Manifesto. In *Proc. of The 5^{tthh} Linux Expo*, 1999.