Towards Co-existing of Linux and Real-Time OSes

Hitoshi Mitake, Tsung-Han Lin, Hiromasa Shimada, Yuki Kinebuchi, Ning Li, Tatsuo Nakajima
Department of Computer Science and Engineering, Waseda University
{mitake, johnny, yukikine, lining, tatsuo}@dcl.info.waseda.ac.jp

Abstract

The capability of real-time resource management in the Linux kernel is dramatically improving due to the effective contribution of the real-time Linux community. However, to develop commercial products cost-effectively, it must be possible to re-use existing real-time applications from other real-time OSes whose OS API differs significantly from the POSIX interface. A virtual machine monitor that executes multiple operating systems simultaneously is a promising solution, but existing virtual machine monitors such as Xen and KVM are hard to used for embedded systems due to their complexities and throughput oriented designs. In this paper, we introduce a lightweight processor abstraction layer named SPUMONE. SPUMONE provides virtual CPUs (vCPUs) for respective guest OSes, and schedules them according to their priorities. In a typical case, SPUMONE schedules Linux with a low priority and an RTOS with a high priority. The important features of SPUMONE are the exploitation of an interrupt prioritizing mechanism and a vCPU migration mechanism that improves real-time capabilities in order to make the virtualization layer more suitable for embedded systems. We also discuss why the traditional virtual machine monitor design is not appropriate for embedded systems, and how the features of SPUMONE allow us to design modern complex embedded systems with less efforts.

1 Introduction

Modern real-time embedded systems like smart phones become highly functional along with the enhancements of CPUs targeting their market. But their functional features introduced significant engineering cost. The main difficulty in the development of such devices comes from the conflicting requirement of them: low latency and high throughput must be established in one system. This requirement is hard to satisfy with existing OSes, because all of them are categorized as either *Real-Time Operating System (RTOS)* or *General Purpose Operating System (GPOS)*. RTOSes, like eCos [2] or TOPPERS[1] [3], are designed and developed for executing real-time tasks such as processing wireless communication protocols. In the typical case, such a task runs periodically for short time. The feature of executing such deadline-sensitive tasks imposes limitation on RTOSes. For example, most RTOSes cannot change the number of tasks dynamically. On the other hand, GPOSes such as Linux, are designed and developed for executing tasks which consist of significant amount of computation. Of course some of them in desktop computers are latency sensitive, to offer a comfortable user experience, but missing a deadline is not fatal for them. The contribution from the real-time Linux community has significantly improved the real-time resource management capability of Linux [7]. However, there is always a tradeoff between satisfying real-time constraints and achieving maximum throughput [8].

In order to develop a modern real-time embedded system which satisfies conflicting requirements, combining multiple OSes on virtual machine monitors can be an effective approach. A virtual machine monitor, e.g. KVM [6], Xen [4] and VMware [5], is traditionally used in the area of data center or desktop computing for executing multiple OS instances in one physical machine. Their capability of executing multiple OSes is also attractive for embedded systems because they make it possible to implement a system which has multiple OS personalities. If there is a virtualization layer which has a capability of executing GPOS and RTOS in one physical machine, developing real-time embedded systems can be simpler.

Armand and Gien [9] presented several requirements for a virtualization layer to be suitable for embedded systems:

[1] TOPPERS is an open source RTOS that offers μITRON interface, and it is used in many Japanese commercial products.

1. It should execute an existing operating system and its supported applications in a virtualized environment, such that modifications required to the operating system are minimized (ideally none), and performance overhead is as low as possible.

2. It should be straightforward to move from one version of an operating system to another one; this is especially important to keep up with frequent Linux evolutions.

3. It should reuse native device drivers from their existing execution environments with no modification.

4. It should support existing legacy often real-time operating systems and their applications while guaranteeing their deterministic real-time behavior.

Unfortunately, there is no open source virtualization layer that satisfies all the above requirements. VirtualLogix[2] VLX [9] is a virtualization layer designed for combining RTOS and GPOS, but it is proprietary software. OKL4 microvisor [12] is a microkernel based virtualization technology for embedded systems, but performs poorly as the nature of microkernels [9]. In addition, we found that there are fatal performance degradation of guest OSes when RTOS and SMP GPOS share a same physical CPU. This performance problem comes from the phenomenon called *Lock Holder Preemption*(LHP) [11]. It is a general phenomenon of virtualization layers, hence a solution for this problem was already proposed. However these existing solutions only focus on the throughput of guest OSes, therefore the virtualization layers that execute RTOSes cannot adopt these solutions. To the best of our knowledge, there is no virtualization layer that can execute RTOS and GPOS on a multicore processor without performance degradation caused by LHP, and is distributed as open source software.

Our laboratory is developing an open source virtualization layer for combining RTOS and Linux on embedded systems that adopt multicore processors, named SPUMONE (Software Processing Unit, Multiplexing ONE into two or more). During the development of this virtualization layer, we faced many difficulties specific to embedded systems. They come from the limitation of hardware resources, the requirement of engineering cost, or scheduling RTOS and SMP GPOS on the same CPU. Because of these difficulties, we believe that virtualization layers for real-time embedded systems should be developed as open source software for incorporating various insights from a wide range of community.

This paper is structured as follows: in Section 2, the detailed motivation of our project is described. Section 3 describes the basic architecture of SPUMONE. Section 4 and Section 5 are catalogs of the problems we encountered. Section 4 describes the difficulties of dealing with real-time virtualization layers which adopt multicore processors. Section 5 describes the method for isolating OSes spatially in embedded systems. Section 6 shows related work and the differences between SPUMONE and them. Finally Section 7 concludes this paper and mentions about future directions of this project.

2 Why Virtualization

This section presents three advantages of using virtualization layers in embedded systems. The first advantage is that control processing can be implemented as application software on RTOS. Embedded systems usually include control processing like mechanical motor control, wireless communication control or chemical control. Using software-based control techniques enables us to adopt a more flexible control strategy, so recent advanced embedded systems contain microprocessors instead of hardware implemented controllers for implementing flexible control strategies. On the other hand, recent embedded systems need to process various information. For example, applications which require significant computation, such as multimedia players and full featured web browsers, are crucial elements of modern smart phones. Therefore, recent embedded systems have to contain both control and information processing functionalities. In traditional embedded systems, dedicated processors are assigned for respective processing. A general purpose processor with sufficient computational capability offers a possibility to combine these multiple processing on a single processor. A virtualization layer can host RTOS and GPOS on one system, therefore this approach requires less hardware controllers and reduces the cost of embedded systems hardware.

The second advantage is that a virtualization layer makes it possible to reuse existing software. Even if the

[2] VirtualLogix, Inc. was acquired by Red Bend Software in Sep. 2010

virtualization layer is based on the para-virtualization technique (which requires the modification of guest OSes), application programs running on the guest OSes do not need to be modified. In a typical case of developing embedded systems, vendors have their own OSes and applications running on them. The virtualization layer can execute such in-house software with standard OS platforms like Symbian or Android. If the in-house software is developed against such a standard platform, it should be modified when a standard platform is replaced. Actually, the standard platform is frequently replaced according to various business reasons. On the other hand, if the in-house software is developed as application programs that run on the vendor specific OSes, porting application programs is not required even if a standard platform is replaced.

The third advantage is the isolation of source code. For example, proprietary device drivers can be mixed with GPL code without license violation. This may solve various business issues when adopting Linux in embedded systems.

3 Basic Architecture

3.1 User-Level Guest OS vs. Kernel Level Guest OS

There are several traditional approaches to execute multiple operating systems on a single processor in order to compose multiple functionalities. Microkernels execute guest OS kernels at the user level. When using microkernels, various privileged instructions, traps and interrupts in the OS kernel need to be virtualized by replacing their code. In addition, since OS kernels executed as user level tasks, application tasks need to communicate with the OS kernel via inter-process communication. Therefore, many parts of the OS need to be modified.

VMMs are another approach to execute multiple OSes. If a processor offers a hardware virtualization support, all instructions that need to be virtualized trigger traps to VMM. This makes it possible to use any OSes without any modification. But if the hardware virtualization support is incomplete, some instructions still need to be complemented by replacing some code to virtualize them.

Most of the processors used for the embedded systems only have two protection levels. So when kernels are located in the privileged level, they are hard to isolate. On the other hand, if the kernels are located in the user level, the kernels need to be modified significantly. Most of embedded system industries prefer not to modify a large amount of the source code of their OSes, so it is desirable to put them in the privileged level. Also, the virtualization of MMU introduces significant overhead if the virtualization is implemented by software. Therefore, we need reorder mechanisms to reduce the engineering cost, to ensure the reliability of the kernels and to exploit some advanced characteristics of multicore processors.

The following three issues are most serious problems, when a guest OS is implemented in the user level.

1. The user level OS implementation requires heavy modification of the kernel.
2. Emulating an interrupt disabling instruction is very expensive if the instruction cannot be replaced.
3. Emulating a device access instruction is very expensive if the instruction cannot be replaced.

In a typical RTOS, both the kernel and application code are executed in the same address space. Embedded systems have dramatically increased their functionalities in every new product. To reduce the development cost, the old version of application code should be reused and extended. The limitation of hardware resources is always the most important issue to reduce the product cost. Therefore, the application code sometimes uses very ad-hoc programming styles. For example, application code running on RTOS usually contains many privileged instructions like interrupt disable/enable instructions to minimize the hardware resources. Also, device drivers may be highly integrated into the application code. Thus, it is very hard to modify these applications to execute at the user level without changing a significant amount of application code, even if their source code is available. Therefore, it is hard to execute the application code and RTOS in the user level without violating the requirements described in Section 1. Therefore, executing RTOS is very hard if the processor does not implement the hardware virtualization support. Even if there is a proper hardware virtualization support, we expect that the performance of RTOS and its application code may be significantly degraded. Our approach chooses to execute both guest OS kernels and a virtualization layer at the same privileged level. This decision makes the modification of OS kernels minimal, and there is no performance degradation by introducing a virtualization layer. However, the following two issues are very serious in the approach.

1. Instructions which disable interrupts have serious impact on the task dispatching latency of RTOS.
2. There is no spatial protection mechanism among OS kernels.

The first issue is serious because replacing interrupt disable instructions is very hard for RTOS and its application code as described above. The second issue is also a big problem because executing guest OS kernels in virtual address spaces requires significant modification on them. SPUMONE proposes a technique presented in Section 4 and Section 5 to overcome the problems.

3.2 SPUMONE: A Multicore Processor Based Virtualization Layer for Embedded Systems

SPUMONE is a thin software layer for multiplexing a single physical CPU (pCPU) core into multiple virtual CPU (vCPU) cores [19, 20]. The current target processor of SPUMONE is the SH4a architecture, which is very similar to the MIPS architecture, and is adopted in various Japanese embedded system products. Also, standard Linux and various RTOSes support this processor. The latest version of SPUMONE runs on a single and multicore SH4a chip. Currently, SMP Linux, TOPPERS [3], and the L4 [12] are running on SPUMONE as a guest OS.

The basic abstraction of SPUMONE is vCPU as depicted in Figure 1. In this example, SPUMONE hosts two guest OSes, Linux and RTOS. Linux has two vCPUs, vCPU0 and vCPU1. vCPU0 is executed by pCPU0 and vCPU1 is executed by pCPU1. RTOS has one vCPU, vCPU2. This is executed by pCPU1. So both of vCPU1 and vCPU2 are executed on pCPU1. Unlike typical microkernels or VMMs, SPUMONE itself and guest OS kernels are executed in the privileged level as mentioned in Section 3.1. Since SPUMONE provides an interface slightly different from the one of the underlying processor, we simply modify the source code of guest OS kernels, a method known as para-virtualization. This means that some privileged instructions should be replaced to *hypervisor calls*, function calls to invoke SPUMONE API, but the number of replacements is very small. Thus, it is very easy to port a new guest OS or to upgrade the version of a guest OS on SPUMONE.

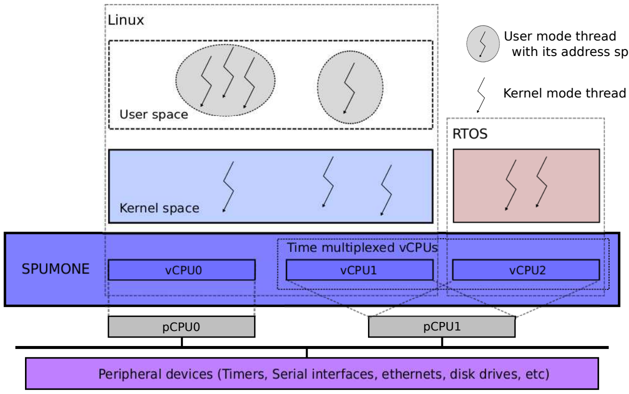

Linux
User space
Kernel space
User mode thread with its address space
Kernel mode thread
RTOS
SPUMONE
Time multiplexed vCPUs
vCPU0
vCPU1
vCPU2
pCPU0
pCPU1
Peripheral devices (Timers, Serial interfaces, ethernets, disk drives, etc)

Figure 1: An Overview of SPUMONE

To spatially protect multiple OSes, if it is necessary, SPUMONE may assume that underlying processors support the mechanisms to protect physical memories used by respective OS like VIRTUS [24]. The approach may be suitable for enhancing the reliability of guest OSes on SPUMONE without significantly increasing overhead. In section 5, we propose an alternative novel approach to use a functionality of multicore processor to realize the spatial protection among guest OSes. The approach does not assume that the processor provides an additional hardware support to spatially isolate guest OSes.

SPUMONE does not virtualize peripheral devices because traditional approaches incur significant overhead that most of embedded systems could not tolerate. In SPUMONE, since device drivers are implemented in the kernel level, they do not need to be modified when the device is not shared by multiple OSes.

Multicore processor version of SPUMONE is designed on the distributed model similar to the Multikernel approach [18]. A dedicated instance of SPUMONE is assigned to each physical core. Therefore, data structures used in SPUMONE need not to be protected by using synchronization mechanisms. This design is chosen in order to eliminate the unpredictable overhead of synchronization among multiple physical cores. This may simplify the design of SPUMONE.

They communicate with each other via the specially allocated 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 or simply reads the data from the shared memory area.

3.2.1 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. Traps are also delivered to SPUMONE first, then are directly forwarded to the currently executing guest OS.

To allow interrupts to be intercepted by SPUMONE, the interrupt entry point of the guest OSes should not be registered to hardware directly. The entry point of each guest OS must notify SPUMONE via a hypervisor call to registering their real vector table. An interrupt is first examined by the interrupt handler of SPUMONE in which the destination vCPU is determined, and the corresponding scheduler is invoked. When the interrupt triggers OS switching, all the registers including MMU state of the current OS are saved into the stack, then the registers in the stack of the previous OS are restored. Finally, the execution is switched to the entry point of the destination OS. The processor initializes the interrupt just as if the real interrupt occurred, so the source code of the OS entry points does not need to be changed.

3.2.2 vCPU Scheduling

Multiple guest OSes run by multiplexing a physical CPU. The execution states of the guest OSes are managed by data structures that we call vCPUs. When switching the execution of vCPUs, all the hardware registers are stored into the corresponding register table of vCPU, and then restored from the table of the next executing vCPU. The mechanism is similar to the process implementation of a typical OS, however the vCPU saves the entire processor state, including the privileged control registers.

The scheduling algorithm of vCPUs is the fixed priority preemptive scheduling. When RTOS and Linux share the same pCPU, the vCPU owned by RTOS would gain a higher priority than the vCPU owned by Linux in order to maintain the real-time responsiveness of RTOS. This means that Linux is executed only when the vCPU of RTOS is in an idle state and has no real-time task to be executed. The process scheduling is left up to OSes so the scheduling model for each OS need not to be changed. Idle RTOS resumes its execution when it receives an interrupt. The interrupt to RTOS should preempt Linux immediately, even if Linux has disabled the execution of its interrupt handlers. The details of this requirement and the solution for it is described in Section 4.1.1.

3.2.3 Modifying Guest OS Kernels

Each guest OS is modified to be aware of the existence of the other guest OSes, because hardware resources other than the processor are not multiplexed by SPUMONE as described below. Thus those are exclusively assigned to each OS by reconfiguring or by modifying their kernels. The following describes how the guest OS kernels are 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 interrupt manager of SPUMONE. Typically this instruction is invoked once during the OS initialization.
- Bootstrap: In addition to the features supported by the single-core SPUMONE, the multicore version provides the virtual reset vector device, which is responsible for resetting the program counter of the vCPU that resides on a different pCPU.
- Physical Memory: A fixed size of 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 code. 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 guest OSes. For these reasons we simply assign a fixed amount of physical memory to each guest 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 physical core to another OS. We prevent the execution of this instruction by replacing it with the hypervisor call of SPUMONE. Typically this instruction is located in a specific part of the kernel, which is fairly easy found and modified.

- 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 the devices can be assigned exclusively to each OS. This assumption is reasonable because, in embedded systems, multiple guest OSes are usually assigned different functionalities and use different physical devices. It usually consists of RTOS and GPOS, where RTOS is used for controlling special purpose peripherals such as a radio transmitter and some digital signal processors, and GPOS is used for controlling generic devices such as various human interaction devices and storage devices. However some devices cannot be assigned exclusively to each OS because both systems need to share them. For instance, the processor we used offers only one interrupt controller. Usually a guest OS needs to clear some of its registers during its initialization. In the case of running on SPUMONE, a guest OS booting after the first one should be careful not to clear or overwrite the settings of the guest OS executed first. For example, we modified the Linux initialization code to preserve the settings done by TOPPERS.

3.2.4 Dynamic Multicore Processor Management

As described in the previous section, SPUMONE enables multiplexing of virtual CPUs on physical CPUs. The mapping between pCPUs and vCPUs is dynamically changed to balance the tradeoffs among real-time constraints, performance and energy consumption. In SPUMONE, a vCPU can be migrated to another core according to the current situation. The mechanism is called the vCPU migration mechanism. In SPUMONE, all kernel images are located in the shared memory. Therefore, the vCPU migration mechanism just moves the register states to manage vCPUs, and the cost of the migration can be reduced significantly. Actually, the round trip time of the vCPU migration in the current version of SPUMONE on the RP1 platform[3] is about 50 μs when a vCPU is moved to anther pCPU and back to the original pCPU.

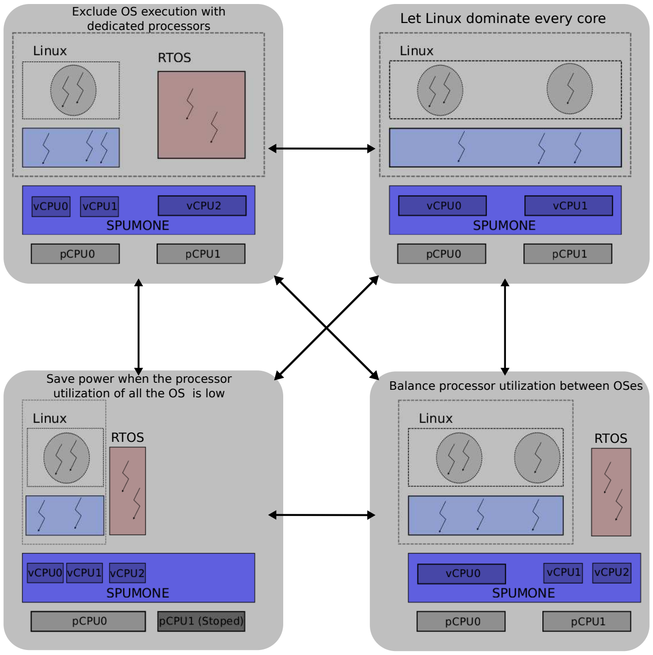

Exclude OS execution with dedicated processors
Linux
RTOS
vCPU0
vCPU1
vCPU2
SPUMONE
pCPU0
pCPU1
Let Linux dominate every core
Linux
vCPU0
vCPU1
SPUMONE
pCPU0
pCPU1
Save power when the processor utilization of all the OS is low
Linux
RTOS
vCPU0
vCPU1
vCPU2
SPUMONE
pCPU0
pCPU1 (Stoped)
Balance processor utilization between OSes
Linux
RTOS
vCPU0
vCPU1
vCPU2
SPUMONE
pCPU0
pCPU1

Figure 2: Dynamically Changing the Mapping Between Virtual CPUs and Physical CPUs

There are several advantages of our approach. The first advantage is to change the mapping between vCPUs and pCPUs to reduce energy consumption. As shown in Figure 2, we assume that a processor offers two p-CPUs. Linux uses two vCPUs and the real-time OS uses one vCPU. When the utilization of RTOS is high, two vCPUs of Linux are mapped on one pCPU (Left Top). When RTOS is stopped, each vCPU of Linux uses a different pCPU (Right Top). Also, one pCPU is used by a vCPU of Linux and another pCPU is shared by Linux and RTOS when the utilization of RTOS is low (Right Below). Finally, when it is necessary to reduce energy consumption, all vCPUs run on one pCPU (Left Below). This approach enables us to use very aggressive policies to balance real-time constraints, performance, and energy consumption.

[3] The RP1 platform is our current hardware platform that contains a multicore processor. The processor has four SH4 CPUs and they are communicated with a shared memory. The platform is developed by Hitach and Renesas.

Configuration	Time	Overhead
Linux Only	68m 5.9s	-
Linux and TOPPERS	69m 3.1s	1.4%

Figure 3: Linux kernel build time

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	-

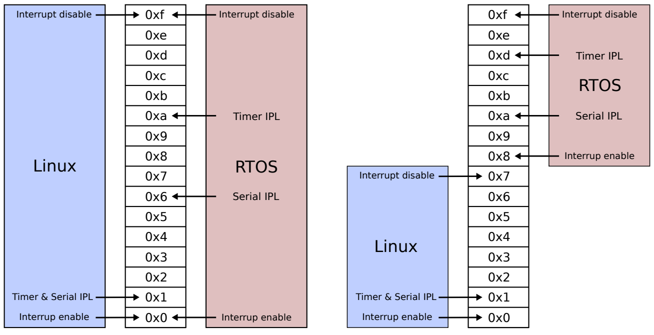
Figure 4: The total number of modified LoC in *.c, *.h, *.S and Makefile

3.2.5 Performance and Engineering Cost

Figure 3 shows the time required to build the Linux kernel on native Linux and modified Linux executed on the top of SPUMONE together with TOPPERS. TOPPERS only receives the timer interrupts every 1ms, and executes no other task. 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 existence of SPUMONE to the system throughput is sufficiently small.

We evaluated the engineering cost of reusing RTOS and GPOS by comparing the number of modified lines of code (LoC) in each OS kernel. Figure 4 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 they are. 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.

The table also shows the modified LoC for RTLinux, RTAI and OK Linux, all of which are previous approaches to support multiple OS environments. Since we could not find RTLinux, RTAI, OK Linux for the SH4a 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 results show that our approach requires only small modifications to the Linux kernel. The result shows that the strategy of SPUMONE, virtualizing processors only, succeeds in reducing the number of modification of guest OSes and to satisfy the requirements described in Section 1.



(a) Without IPL separation

(b) With IPL separation

Figure 5: Separating Interrupt Priorities Between Guest OSes

4 Real-Time Resource Management in SPUMONE

4.1 Reducing RTOS Dispatch Latency

In order to minimize the dispatch latency of RTOS tasks during concurrent activities of Linux on a single device, we propose the following two techniques in SPUMONE.

4.1.1 Interrupt Priority Level Separation

The first technique is to replace the interrupt enabling and disabling instructions with the hypervisor calls. A typical OS disables all interrupt sources when disabling interrupts for the atomic execution. For example, `local_irq_enable()` of Linux enables all interrupt and `local_irq_disable()` disables all interrupt. On the other hand, our approach leverages the interrupt priority mechanism of the processor. The SH4a processor architecture provides 16 interrupt priority levels (IPLs). We assign the higher half of the IPLs to RTOS and the lower half to Linux as shown in Figure 5. When Linux tries to block the interrupts, it modifies its interrupt mask to the middle priority. RTOS may therefore preempt Linux even if it is disabling the interrupts. On the other hand, when RTOS is running, the interrupts for

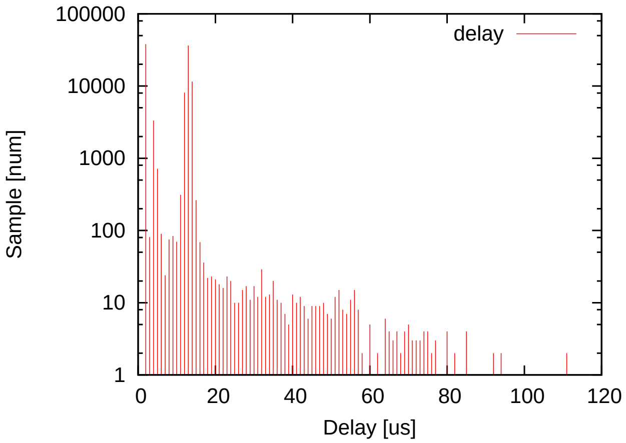


Figure 6: Interrupt dispatch latency of TOPPERS without IPL separation

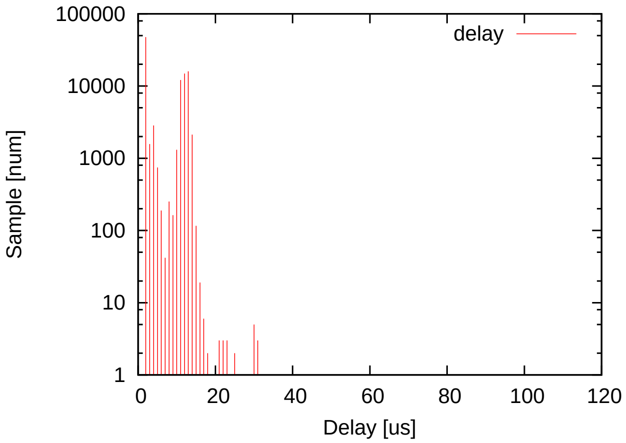


Figure 7: Interrupt dispatch latency of TOPPERS with IPL separation

Linux are blocked by the processor. These blocked interrupts could be delivered immediately when Linux is dispatched.

The instructions for enabling and disabling interrupts are typically provided by the kernel internal API like `local_irq_enable()` and `local_irq_disable()`. They are typically coded as inline functions or macros in the kernel source code. For Linux, we replace `local_irq_enable()` with the hypervisor call which enables entire level of interrupts and `local_irq_disable()` with the other hypervisor call which disables the lower priority interrupts. For RTOS, we replace the API for interrupt enabling with the hypervisor call enabling only high priority interrupts, and the API for interrupt disabling with the other hypervisor call disabling the entire level of interrupts. Therefore, interrupts assigned to RTOS are immediately delivered to RTOS, while interrupts assigned to Linux are blocked during execution of the RTOS. Figure 5 shows the interrupt priority levels assignment for each OS, which we used in the evaluation environment.

Figures 6 and 7 show the task dispatch latency of TOPPERS under two configurations of SPUMONE. In Figure 6, the evaluation result of SPUMONE without the IPL separation executing Linux and TOPPERS is depicted. Linux executes `write()` on the file stored on a Compact Flash card repeatedly and TOPPERS measures the task dispatch latency of the interrupts from time management unit. In Figure 7, the result of SPUMONE with the IPL separation is shown. The guest OSes and their workloads are the same as the condition used in a case when the IPL separation is not used.

As these results show, the workload of Linux heavily interference with the task dispatch latency of RTOS if the IPL separation is not configured. Therefore we can say that separating IPL is an effective method to guarantee the low interrupt dispatch latency of RTOS.

However, the approach assumes that all activities in TOPPERS are processed at the higher priority than the activities of Linux. The current version of Linux is improving real-time capabilities. So, in the near future, some applications that requires to satisfy real-time constraints will be developed on Linux. In this case, the approach described here cannot be used. Also, the approach increases the number of modifications of Linux. It is desirable not to replace interrupt enable/disable instructions in terms of the engineering cost. Therefore, we have developed an alternative method described in the next section.

4.1.2 Reducing Task Dispatching Latency with vCPU migration

The second technique is based on the vCPU migration mechanism introduced in Section 3.2.4. The first technique, replacing API for interrupt enabling/disabling requires slight but non-trivial modification of Linux. In addition, the technique may not work correctly when the device drivers or kernel modules are programmed in a bad manner, which enable or disable interrupts with a non-standard way. The second technique exploits the vCPU migration mechanism. Under this technique, SPUMONE migrates a vCPU, which is assigned to Linux and shares the same pCPU with the vCPU of RTOS, to another pCPU when it traps into kernel mode, or when interrupts are received. In this way, only the user level code of Linux is executed concurrently on the shared pCPU, which will never change the priority levels. Therefore, RTOS may preempt Linux immediately without separating IPL used in the first technique.

4.2 Increasing the Throughput of SMP Linux

Generally speaking, porting OSes to virtualization layers produces semantic gap because the assumptions which guest OSes rely on may not be preserved. For example, OSes assume that they dominate CPU, memory, and storage. In the ordinary environment where OSes run directly on the real hardware, this assumption is true. But when virtualization layers execute guest OSes, this assumption is no longer held. CPU and memory are shared by multiple OSes.

The semantic gap produced by virtualization layers can cause some new problems. One of the typical problems is called the Lock Holder Preemption (LHP) problem [11].

The LHP problem occurs when the vCPU of the guest OS is preempted by the virtualization layer during the execution of critical sections protected by mutex based on busy waiting (e.g. `spinlock_t`, `rwlock_t` in Linux). Figure 8 depicts the typical scenario of LHP in SPUMONE. On SPUMONE, the execution of vCPU2 belonging to RTOS is started immediately even if vCPU1 executing Linux is currently running on the same processor, because the activities of RTOS are scheduled at a higher priority than the activities in Linux. Let us assume that the execution of the Linux kernel is preempted while the kernel keeps a lock. In this case, other vCPUs owned by Linux and running on other pCPUs may wait to acquire the lock via busy waiting.

4.2.1 Existing Solutions of LHP

This performance degradation problem caused by LHP is a general one of every virtualization layer. So, there are existing solutions for solving the problem. Uhlig, et al. pointed this problem [11]. They also introduced the methods to avoid the problem. The method is named as *Delayed Preemption Mechanism (DPM)*. DPM is suitable for a virtualization layer based on the para-virtualization technology because it does not waste CPU time and can be implemented with a less effort. However, this solution increases the dispatch latency of guest OSes, making it unsuitable for embedded systems that need to satisfy real-time constraints.

VMware ESX employs a scheduling algorithm called co-scheduling [15] in its vCPU scheduler [16]. This solution wastes lots of CPU time. VMware ESX employs the technique because it is the full-virtualization technique. Also, it does not assume to execute multiple vCPUs for a guest OS on one pCPU. Sukwong and Kim introduced the improved co-scheduling algorithm named *balance scheduling* and implemented it on KVM [17].

Wells, et al. introduced a hardware based solution called *spin detection buffer (SDB)* for detecting meaningless spin of vCPUs produced by LHP [13]. They found that the execution pattern can be distinguished when a CPU is spinning before acquiring a lock. SDB inspects the number of store instructions and counts the number of updated memory address if a thread is executed in kernel mode. If the number of counted addresses does not exceed the threshold (they set it as 1024), SDB judges the thread is spinning in vain. This hardware information can be used by a virtualization layer to avoid the LHP problem.

Friebel and Biemueller introduced a method for avoiding LHP on Xen [14]. Respective threads in the guest OSes count the number of spinning on a busy wait mutex. When the count exceeds the threshold, the spinning thread invokes the hypervisor call in order to switch to another vCPU.

4.2.2 Solving LHP in SPUMONE

The methods described in Section 4.2.1 improve the throughput of SMP Linux on traditional VMMs. But all of them assume that there are no real-time activities.

In this section, we propose a new method for avoiding LHP. In our approach, the vCPU of Linux, which shares pCPU with the vCPU of RTOS, is migrated to another pCPU when an interrupt for RTOS is received. Then, it returns to the original pCPU when RTOS yields pCPU and becomes idle. When two vCPUs for Linux are executed on the same pCPU, they also cause the

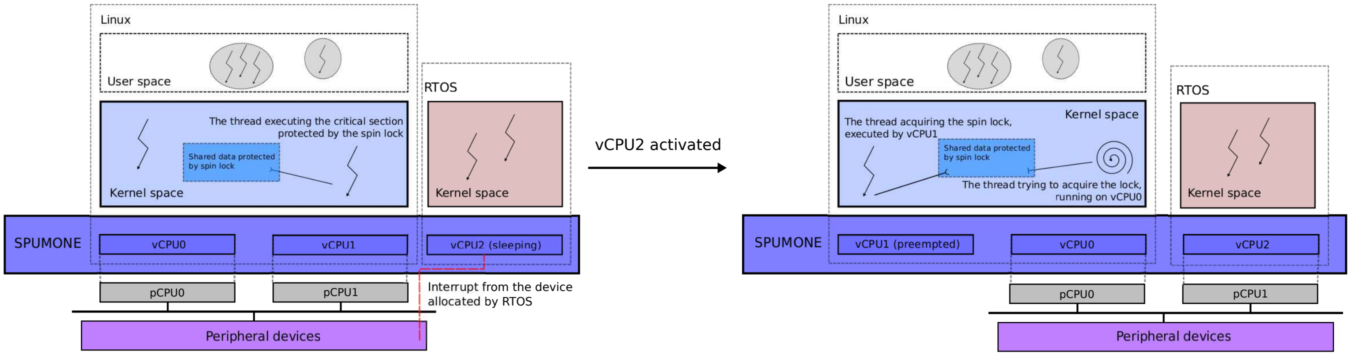

Linux
User space
The thread executing the critical section protected by the spin lock
Shared data protected by spin lock
Kernel space
RTOS
Kernel space
SPUMONE
vCPU0
vCPU1
vCPU2 (sleeping)
pCPU0
pCPU1
Interrupt from the device allocated by RTOS
Peripheral devices
vCPU2 activated
Linux
User space
The thread acquiring the spin lock, executed by vCPU1
Kernel space
Shared data protected by spin lock
The thread trying to acquire the lock, running on vCPU0
RTOS
Kernel space
SPUMONE
vCPU1 (preempted)
vCPU0
vCPU2
pCPU0
pCPU1
Peripheral devices

Figure 8: Typical example of Lock Holder Preemption in SPUMONE

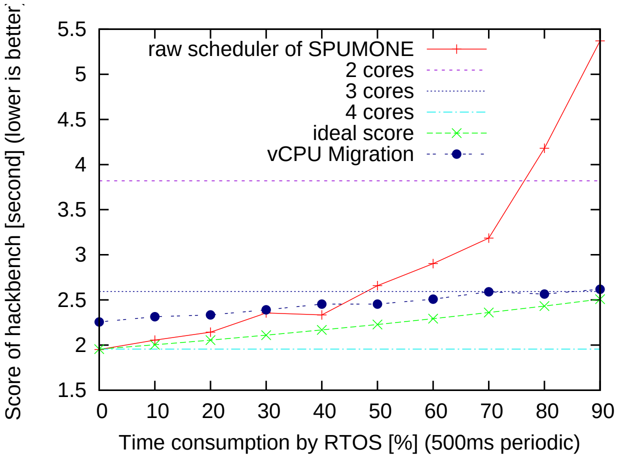


Figure 9: Result of hackbench on Various Configuration

LHP problem. But, in this case, we assume that the delayed preemption mechanism can be used since Linux does not have real-time activities. The vCPU migration mechanism is similar to the thread migration in normal OSes, but in the case of SPUMONE, interrupt assignments have to be reconfigured because peripherals devices are not virtualized. In our evaluation environment, timer interrupts and ICI should be taken into account. Let us assume that vCPU0 is migrated from pCPU0 to pCPU1 while executing an activity on vCPU1. The timer device raising interrupts periodically for vCPU0 on pCPU1 should be stopped before the vCPU migration. Then, the timer device on pCPU1 should be multiplexed for both vCPU0 and vCPU1. Also, ICI for vCPU0 on pCPU0 should be forwarded to vCPU0.

Figure 9 shows the hackbench score on various configurations of SPUMONE. In this evaluation environment, four pCPUs execute five vCPUs. Therefore two vCPU share one pCPU. One vCPU belongs to TOPPERS and four vCPUs belong to Linux. TOPPERS executes a task which consumes CPU time in the 500ms period. Linux executes hackbench for measuring its throughput. The X axis means the CPU consumption rate of the task on TOPPERS, and the Y axis means the score of hackbench. Three horizontal lines describe the score of hackbench under the case that Linux dominates pCPUs.

The line indicated as “ideal score” describes the score which we expected at first. When RTOS consumes the time of $f (0 \leq f < 1)$ on one pCPU, Linux should exploit the rest of CPU resources: $4 - f$ (When $f = 1$, which means RTOS never yields pCPU, the rest of CPU resources is not equal to 3. Because this is the same as the situation when 1 CPU stops execution suddenly from the perspective of Linux). The line of the ideal score is calculated as: $I(f) = \frac{S_1}{4-f}$ where f means the CPU consumption rate of RTOS and S_1 means the score of Linux dominating one core. hackbench has enough parallelism, therefore we predicted that the score might be linear according to the CPU consumption rate of RTOS.

The score actually measured is presented as the line indicated as “raw scheduler of SPUMONE”. We noticed that the rapid degradation of the performance is caused by LHP. So we designed and implemented the new method described above. The score measured when using the new method is described as the line indicated as “vCPU migration”. This score is still worse than the ideal score, but it sufficiently utilizes the CPU resource because it is better than or nearly equal to the case when Linux dominates three cores.

Current score when using our new method is still worse than the ideal score, so more optimization or better vCPU scheduling policy is required. We are planning to apply the method described in Section 4.1.2. In modern systems, mutexes based on busy wait mechanism are only used in kernel space. Therefore if the vCPU of Linux, shareing pCPU with the vCPU of RTOS, is migrated to another pCPU when the thread invokes system calls or the interrupts for Linux rises, LHP can be avoided.

4.3 Real-Time Task Aware Scheduler

One of the ongoing projects of SPUMONE, we plan to use these additional resources to further improve the real-time capability of guest OSes, especially Linux, by dynamically scheduling the vCPU of the guest OSes on top of the SPUMONE. In the original design strategy of SPUMONE, we gave a high priority to the vCPU of RTOS which is higher than the priority of the vCPU of Linux. But this is not always the case; there might exist some real-time processes that have quicker response time requirements than the RTOS processes. In this situation, we can mark one of the vCPUs of Linux as rt-vCPU and schedule it against vCPU of RTOS. When the priority of this rt-vCPU is higher than that of the vCPU of RTOS, it can gain the control of the pCPU, but simultaneously, because we have some other pCPU in multicore system, we can migrate the vCPU of RTOS to another core and compete with other vCPUs, so the overall performance will not be harmed too much. But the overhead of this migration operation has to be carefully taken care of.

5 Offering Spatial Protection in SPUMONE

As described in Section 3, SPUMONE locates guest OS kernels and SPUMONE in the same privileged level. However, the Linux kernel might contain security holes because of its huge source code, and there are possibilities to infect the Linux kernel. For increasing the reliability of the entire system, a virtualization layer should offer mechanisms to protect a virtualization later and coexisting RTOSes. In traditional approaches, strict memory isolation is used for the protection, but our approach cannot rely on the traditional solution because it is too expensive for typical embedded systems. SPUMONE offers two mechanism to increase the reliability of the entire system. In the following sections, we will explain the mechanisms in detail.

5.1 Protecting SPUMONE and RTOS

In the virtualization environment of SPUMONE, guest OS kernels are running side by side with SPUMONE in the most privileged level. This means that these kernels and SPUMONE could be affected by one and another. In order to further improve the security of the system, we try to better protect these kernels without implementing too much functionality in SPUMONE. We did so by taking advantage of the distributed design of SPUMONE in the multicore platform [21]. Multicore design of SPUMONE is different from traditional VMMs in that each physical core has its own dedicated virtualization layer instance, while the traditional ones have only one instance across all available physical cores. We then simply install each SPUMONE instance into the local memory area of each physical core. Because the content of local memory is only accessible from its own physical core, the attacks or intrusions from the other cores are therefore prohibited as shown in Figure 10. This also means that the attacks will not propagate. When one part of the system is broken and tries to affect others, it will not make it. So the remaining part of the systems can operate normally.

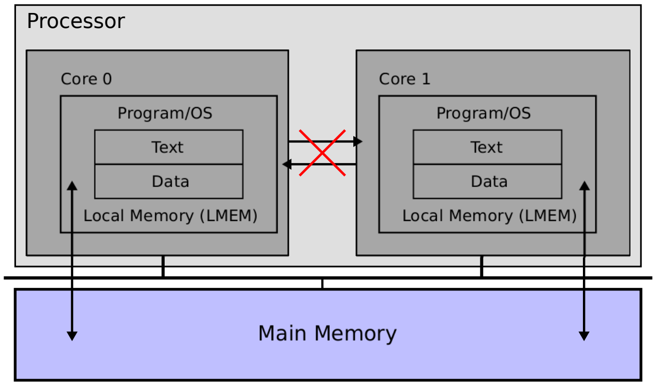


Figure 10: Separated Core-Local Memory and Its Application for Security

5.1.1 Core-Local Memory

Let us assume that two OS kernels running on top of a dual-core processor where each core has an independent core-local memory. If the following assumptions are satisfied, an OS kernel is protected from others.

1. The size of an OS kernel is small enough to fit in core-local memory.
2. Each core should be restricted to reset other core where the reset cleans up the content of the core-local memory.

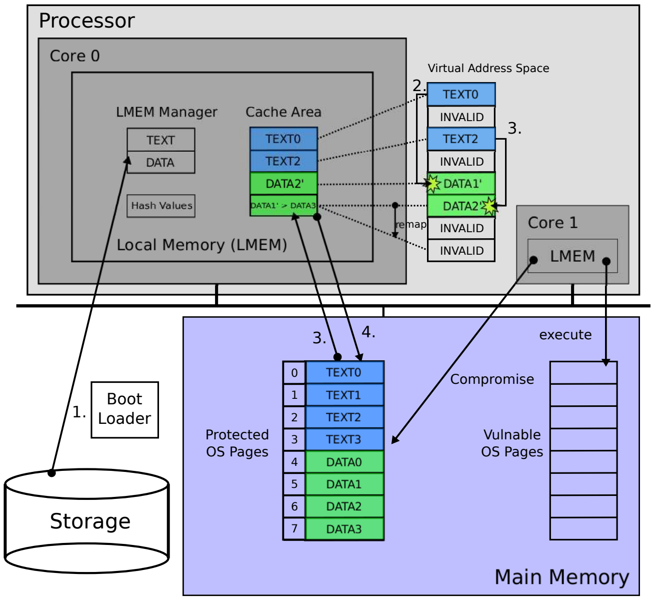


Figure 11: An Overview of Hash-Based Integrity Management

3. The boot image of an OS kernel should not be infected, and a secure boot loader can load the kernel image in the shared memory correctly.

4. Each core should be restricted to access I/O devices. I/O devices that are managed by a core should not be accessed from other cores.

5.1.2 Hash-Based Integrity Management

The problem of the solution presented in the previous section is the size of core-local memory. Currently they are a few hundred KBs. It is too small to load a modern RTOS. In order to virtually extend the size of a local memory, we propose a hash-based integrity management mechanism assisted by the core-local memory protection. The original kernel image is stored in the shared main memory, and a subset of the kernel image is copied to the core-local memory before execution by the core. When a part of the kernel image is loaded in the core-local memory, this part is verified every time to make sure that it is not corrupted or infected.

We present how the hash-based integrity management works in Figure 11. The page allocation in a core-local memory and the calculation of cryptographic hash values are managed by the local memory (LMEM) manager that resides permanently in the core-local memory. An OS kernel image that can be protected from other OS kernels is called a protected OS (pOS). An OS kernel that may be infected by malicious activities is called a vulnerable OS (vOS). pOS and vOS must run on different cores.

1. The boot loader loads the LMEM manager into the core-local memory. The OS kernel images of pOS and vOS are loaded at the same time into the main memory. LMEM calculates the hash value of each page of pOS, and stores it in a hash table also located in the core-local memory. The manager loads a memory page that contains the entry point of pOS into the core-local memory. Then the other core may start to execute vOS.

2. The pages of pOS are mapped in a virtual address space, and a page table for managing the virtual address space should be in the core-local memory. When the size of the page table is bigger than the size of the core-local memory, LMEM can swap out unused page tables to the shared memory. LMEM also manages the hash table for maintaining the integrity of the swapped page tables. LMEM manages page faults when the page table does not contain a corresponding page table entry.

3. When LMEM handles a page fault, a corresponding page is copied from the shared memory to the core-local memory. LMEM calculates the hash value of the page and compares it with the precalculated value stored in the core-local memory. A mismatch of the hash value means that the image of pOS in the shared memory is corrupted. If there is no mismatch, the page fault is correctly completed and the execution of pOS is resumed.

4. When there is no space available in the core-local memory, LMEM swaps out some pages to the shared memory. LMEM checks whether the page is updated or not, and if it is updated, LMEM recalculates the hash value of the page and updates the hash table entries. The pages will be used for loading other pages.

In this approach, the image of pOS in the shared memory may be corrupted by vOS. Our current policy is to restart pOS by reloading a new undamaged kernel image by a secure loader. We are also considering a technique to protect a kernel image by using a memory error correction technique and an encryption technique.

6 Related Work

Virtualization technologies are already used in the area of the desktop and data center computing today [4, 5, 6]. It is also becoming a strong technique for constructing real-time embedded systems because of an enhancement of processors targeting embedded systems market.

RTLinux [22] is a well known hard real-time solution for Linux, but it is also known with its patent problems. RTAI [23] is another real-time extension for Linux and is distributed as free software, but it requires significant modification of the source code of Linux.

KVM for ARM [10] is a KVM based lightweight paravirtualization technology for ARM processors. This might be a strong candidate of virtualization technology for real-time embedded systems because it only requires automated modification of the guest Linux.

OKL4 [12] is a hypervisor based on the micro-kernel design. Armand and Gien introduced the poorness of its performance come from the design of micro-kernel [9]. VirtualLogix VLX [9] is a practical designed VMM for real-time embedded systems.

And in our best knowledge, none of them can handle LHP on multicore processors while guaranteeing real-time responsiveness when the guest OSes have asymmetrical priorities and roles.

7 Conclusion and Future Directions

Before concluding our paper, we would like to share our experiences with the difficulties to promote open source software in Japanese embedded system industries.

We have been discussing various aspects of open source software with embedded system industries for a long time. We found that a lot of people in industries who are working on open source software are aware of the the merits. Especially, asking questions to communities is very useful to find good solutions to problems. However, their bosses who were hardware engineers before do not understand the merits because in their cases, the solutions should be solved by themselves inside of their industries. The cultural gaps between the generations inside industries becomes one of the biggest obstruction to work with open source communities.

Now, we already know that social networks have significantly strong power on sharing knowledge. Open source communities are kind of social networks to share knowledge about open source software, and engineers who ask questions to communities also need to answer question of other people in the communities, but it sometimes too difficult to make time for discussing open source communities while they are working.

Also, it is not easy that embedded system industries contribute their software on open source communities because they sometimes use the old version of software. Because the modification of the old version is not easy to be integrated into the current version of the open source software.

In this paper, we introduced SPUMONE that is a virtualization layer for multicore processor based embedded systems. We described an overview of SPUMONE and showed how SPUMONE reduces the RTOS dispatch latency and protects SPUMONE and RTOS from malicious attacks on the Linux kernel. Currently, we are preparing to distribute SPUMONE as open source software.

References

[1] Tatsuo Nakajima, Yuki Kinebuchi, Hiromasa Shimada, Alexandre Courbot, Tsung-Han Lin. Temporal and Spatial Isolation in a Virtualization Layer for Multi-core Processor based Information Appliances. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, 2011.

[2] eCos. http://ecos.sourceware.org

[3] TOPPERS Project. http://www.toppers.jp/en/index.html

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[5] VMware. http://www.vmware.com

[6] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin Qumranet, Anthony Liguori. kvm: the Kernel-based Virtual Machine. In *Proceedings of Ottawa Linux Symposium*, 2007.

[7] Ingo Molnar. RT-patch. http://www.kernel.org/pub/linux/kernel/projects/rt/

[8] Paul E McKenney. 'Real Time' vs. 'Real Fast': How to Choose? In *Proceedings of the Ottawa Linux Symposium*, 2008.

[9] François Armand and Michel Gien. A Practical Look at Micro-Kernels and Virtual Machine Monitors. In *Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference*, 2009.

[10] Christoffer Dall and Jason Nieh. KVM for ARM. In *Proceedings of Ottawa Linux Symposium*, 2010.

[11] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards Scalable Multiprocessor Virtual Machines. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*

[12] Open Kernel Labs. OKL4 Microvisor. http://www.ok-labs.com/products/okl4-microvisor

[13] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Hardware Support for Spin Management in Overcommitted Virtual Machines. In Proc. *of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT-2006),* Sept. 2006, Seattle, WA

[14] Thomas Friebel and Sebastian Biemueller. How to Deal with Lock Holder Preemption. http://www.amd64.org/fileadmin/user_upload/pub/2008-Friebel-LHP-GI_OS.pdf

[15] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. *Proceedings of Third International Conference on Distributed Computing Systems, 1982*

[16] VMware, Inc. VMware vSphere(TM) 4: The CPU Scheduler. in VMware(R) ESX(TM) 4 http://www.vmware.com/files/pdf/perf-vsphere-cpu_scheduler.pdf

[17] Orathai Sukwong and Hyong S. Kim. Is Co-scheduling Too Expensive for SMP VMs? In *Proceedings of the ACM European conference on Computer systems*, 2011.

[18] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris Rebecca Isaacs, Simon Peter Timothy Roscoe, Adiran Schüpbach, Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009.

[19] Yuki Kinebuchi, Takushi Morita, Kazuo Makijima, Midori Sugaya, Tatsuo Nakajima. Constructing a Multi-OS Platform with Minimal Engineering Cost. In *proceedings of Analysis, Architectures and Modelling of Embedded Systems*, 2009.

[20] Tatsuo Nakajima, Yuki Kinebuchi, Alexandre Courbot, Hiromasa Shimada, Tsung-Han Lin, Hitoshi Mitake. Composition kernel: a multi-core processor virtualization layer for rich functional smart products. In *Proceedings of the 8th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*, 2010.

[21] Tsung-Han Lin, Yuki Kinebuchi, Alexandre Courbot, Hiromasa Shimada, Takushi Morita, Hitoshi Mitake, Chen-Yi Lee and Tatsuo Nakajima. Hardware-assisted Reliability Enhancement for Embedded Multicore Virtualization Design. In *the Proceedings of 14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing*, 2011.

[22] Victor Yodaiken. The RTLinux Manifesto. In *the Proceedings of the 5th Linux Expo, March 1999, in Raleigh North Carolina*

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

[24] Hiroaki Inoue, Junji Sakai, Masato Edahiro. Processor virtualization for secure mobile terminals. In *ACM Transactions on Design Automation of Electronic Systems*, Volume 13 Issue 3, July 2008.