Light-Weighted Virtualization Layer for Multicore Processor-Based Embedded Systems

Hitoshi Mitake, Hiromasa Shimada, Tsung-Han Lin, Ning Li, Yuki Kinebuchi,
Chen-Yi Lee, Daisuke Yamaguchi, Takumi Yajima, Tatsuo Nakajima

Department of Computer Science and Engineering
Waseda University
{mitake,h-shimada,johnny,lining,yukikine,victor,no-trick-in-go,takumi-yajima,tatsuo}@dcl.info.waseda.ac.jp

Abstract

The real-time resource management in the Linux kernel is dramatically improving due to the effective contribution of the real-time Linux community. However, reusing existing real-time applications in embedded systems is required to develop commercial products without significantly increasing their cost because existing real-time applications run on real-time OSes whose OS API is significantly different 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 be used for embedded systems due to their complexities and throughput oriented designs. In this paper, we introduce a lightweight processor abstraction layer named *vlk*. *vlk* provides virtual CPUs (vCPUs) for respective guest OSes, and schedules them according to their priorities. In a typical case, *vlk* schedules Linux with a low priority and an RTOS with a high priority. Two important features of *vlk* are 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 *vlk* 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 or TOPPERS[1], are designed and developed for executing real-time tasks such as processing wireless communication protocols. In a typical case, these tasks run periodically for short time. The feature of executing such deadline sensitive tasks relies on the limitation to RTOSes. For example, most RTOSes cannot change the number of tasks dynamically. On the other hand, GPOSes, like 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 for offering the comfortable experience to users, but missing deadlines is not fatal for them. The contribution from the real-time Linux community has significantly improved the real-time resource management capability of Linux [16]. However, there is always a tradeoff between satisfying real-time constraints and achieving maximum throughput [9].

In order to develop such a modern real-time embedded system which needs to satisfy conflicting requirements, combining multiple OSes on a virtual machine monitor can be an effective approach. Virtual machine monitors, e.g. KVM [7], Xen [4] and VMware [18], are 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 the 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, development of real-time embedded systems can be simpler.

In [2], Armand and Gien presented several requirements for a virtualization layer to be suitable for embedded systems:

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.

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

Unfortunately, there is no open source virtualization layer that has a capability to satisfy above all requirements. VirtualLogix[2] VLX [2] is a virtualization layer designed for combining RTOS and GPOS, but it is proprietary software. OKL4 microvisor [17] is a microkernel based virtualization technology for embedded systems, but performs poorly as the nature of microkernels [2]. In addition, we found that there is fatal performance degradation of guest OSes when RTOS and SMP GPOS share the same physical CPU. This performance problem comes from the phenomenon called *Lock Holder Preemption*(LHP) [13]. 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 *vlk* (vCPU Layer in Kernel), a forked project from our original project named SPUMONE. 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 *vlk*. Section 4 describes the difficulties of dealing with real-time virtualization layers which adopt multicore processors. Finally Section 5 concludes this paper and mentions about future directions of this project.

2. Why Virtualization

This section presents four advantages to use the virtualization layer 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 amount of computations like multimedia players and full featured web browsers are crucial ones 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 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, a vendors has their own OSes and the applications runs 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 as software that depends on 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.

The fourth advantage is the isolation of mutual exclusions between guest OSes. Yodaiken explained that the priority inheritance mechanism are not suitable for designing real-time systems because of its high overhead and complexity [19]. In short, Yodaiken concluded that making critical sections fast and short is the essential contribution for real-time responsiveness. McKenney also showed that the priority inheritance mechanism implemented in the rt patch of Linux produces the overhead which affects throughput performance [9]. In general, the priority inheritance mechanism radically contributes to the low latency with preserving original semantics of mutexes, even if the OS contains various length of critical sections. But it sacrifices the throughput performance as the trade-off.

Of course, the rt patch makes writing soft real-time applications with POSIX APIs possible and this feature is very important and useful especially in the area of enterprise computing. But real-time applications in the embedded systems area do not require such a rich functional APIs like POSIX.

With hosting RTOS and GPOS, the isolation of mutual exclusions can be established because the OSes have their own mutual exclusion mechanisms. The threads of the RTOS never acquire the mutex for protecting critical sections of the GPOS, and vice versa. Of course the mutexes for protecting data structures and critical sections used for inter OS communication must be implemented, but the mutexes and the critical sections they protect are easier for certifying shortness than the one in modern GPOS like Linux.

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 are to be 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

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

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. For reducing 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 use very adhoc 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 application code to be executed 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. *vlk* proposes a technique to solve the first issue presented in Section 4 and the second issue is presented in [10].

3.2 *vlk*: A Multicore Processor based Virtualization Layer for Embedded Systems

vlk is a thin software layer for multiplexing a single physical CPU(pCPU) core into multiple virtual CPU(vCPU) cores. The current target processor of *vlk* 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 *vlk* runs on a single and multicore SH4a chip. Currently, SMP Linux, TOPPERS, and the L4 [17] are running on *vlk* as a guest OS.

The basic abstraction of *vlk* is vCPU as depicted in Figure 1. In the example of this figure, *vlk* hosts two guest OSes, Linux and

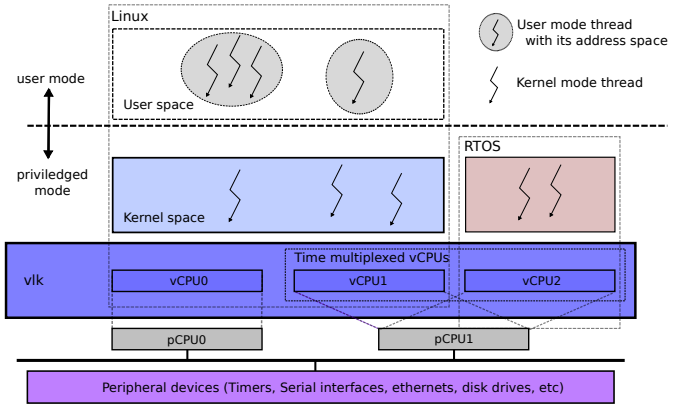

Figure 1. An Overview of *vlk*

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, *vlk* itself and guest OS kernels are executed in the privileged level as mentioned in Section 3.1. Since *vlk* 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 *vlk* 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 *vlk*.

vlk does not virtualize peripheral devices because traditional approaches incur significant overhead that most of embedded systems could not tolerate. In *vlk*, 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.

3.2.1 Interrupt/Trap Delivery

Interrupt virtualization is a key feature of *vlk*. Interrupts are intercepted by *vlk* before they are delivered to each guest OS. When *vlk* 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 *vlk* first, then are directly forwarded to the currently executing guest OS.

For intercepting interrupts by *vlk*, the interrupt entry point of the guest OSes should not be registered to hardware directly. The entry point of each guest OS is notified to *vlk* via a hypervisor call for registering their real vector table. An interrupt is first examined by the interrupt handler of *vlk* in which the destination vCPU is decided, 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 needs 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 is disabling 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 *vlk* 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 *vlk*.

- 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 *vlk*. Typically this instruction is invoked once during the OS initialization.
- Bootstrap: In addition to the features supported by the single-core *vlk*, 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 *vlk*. 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 *vlk* 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 *vlk*, the 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.

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

Figure 2. Linux kernel build time

OS(Linux version)	Added LoC	Removed LoC
Linux/*vlk*(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 3. The total number of modified LoC in *.c, *.h, *.S and Makefile

3.2.4 Dynamic Multicore Processor Management

As described in the previous section, *vlk* enables to multiplex multiple 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 *vlk*, a vCPU can be migrated to another core according to the current situation. The mechanism is called the vCPU migration mechanism. In *vlk*, 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 *vlk* on the RP1 platform[3] is about 50 μ when a vCPU is move to anther pCPU and back to the original pCPU.

3.2.5 Performance and Engineering Cost

Figure 2 shows the time required to build the Linux kernel on native Linux and modified Linux executed on the top of *vlk* together with TOPPERS. TOPPERS only receives the timer interrupts every 1ms, and executes no other task. The result shows that *vlk* 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 *vlk* 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 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 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 that are previous approaches to support the 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 it is clear that our approach

[3] The RP1 platform is our current hardware platform that contains a multicore processor. The processor has four SH4a CPUs and they are communicated with a shared memory. The platform is developed by Hitach and Renesas.

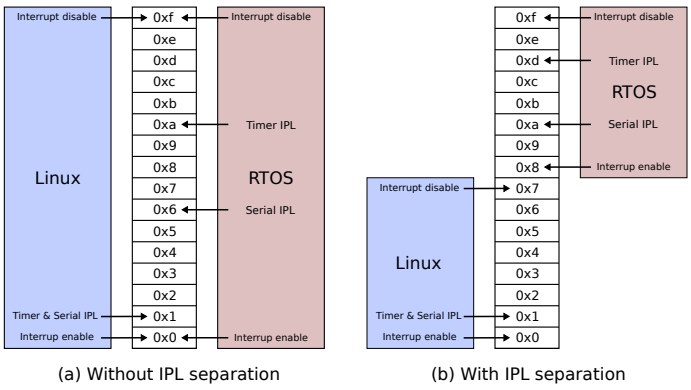

Figure 4. Separating Interrupt Priorities Between Guest OSes

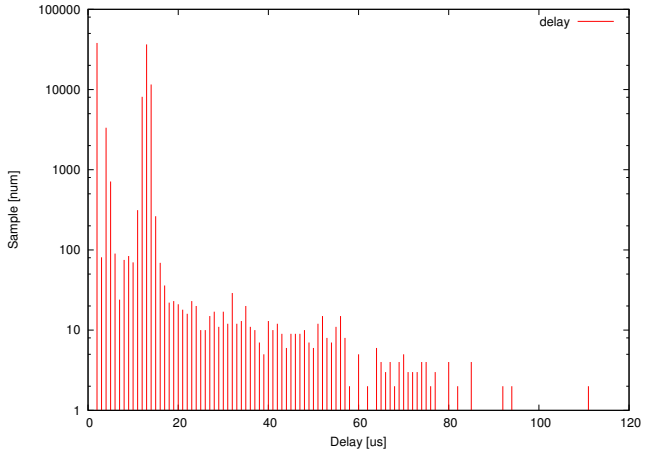

Figure 5. Interrupt dispatch latency of TOPPERS without IPL separation

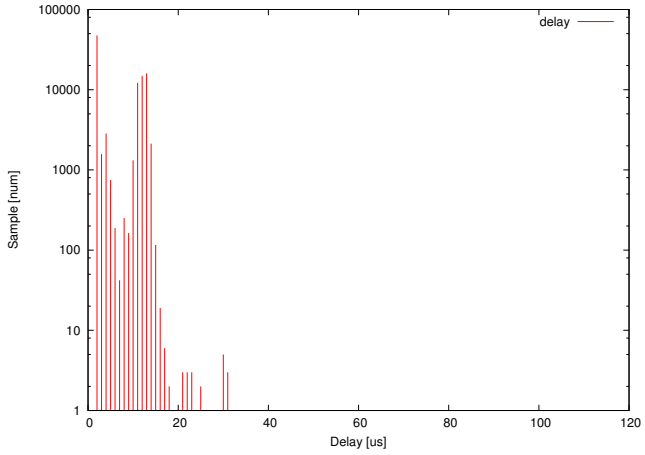

Figure 6. Interrupt dispatch latency of TOPPERS with IPL separation

requires significantly small modifications to the Linux kernel. The result shows that the strategy of *vlk*, virtualizing processors only, succeed to reduce the amount of modification of guest OSes and to satisfy the requirements described in Section 1.

4. Real-Time Resource Management in *vlk*

4.1 Reducing RTOS Dispatch Latency

In order to minimize the dispatch latency of RTOS tasks although the activities of Linux running concurrently on a single device, we propose the following two techniques in *vlk*.

4.1.1 Interrupt Priority Level Separation

The first technique is replacing 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 prioritize 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 4. 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 Linux are blocked by the processor. These blocked interrupts could be delivered immediately when Linux is dispatched.

The instructions enabling and disabling interrupts are typically provided as 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 another hypervisor call which disables the lower prioritized 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, and the interrupts assigned to Linux are blocked during the execution of the RTOS. Figure 4 shows the interrupt priority levels assignment for each OS, which we used in the evaluation environment.

In Figure 5 and Figure 6, the results of task dispatch latency of TOPPERS under two configurations of *vlk* are depicted. In Figure 5, the evaluation result of *vlk* 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 6, the result of *vlk* with the IPL separation is described. 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 interfere 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 in the section cannot be used. Also, the approach increases an amount of the modification of Linux, and increases the engineering cost if it needs to replace interrupt enable/disable instructions. 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 slightly but certain 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, *vlk* 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 the kernel mode or 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 ordinal environment where OSes run on the real hardware directly 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 [13].

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 7 depicts the typical scenario of LHP in *vlk*. On *vlk*, the execution of vCPU2 belongs 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 the 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 for acquiring 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 [13]. 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, then it is not suitable for embedded systems that need to take into account satisfying real-time constraints.

VMware ESX employs the scheduling algorithm called co-scheduling [11] in its vCPU scheduler [15]. 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 [12].

Wells, et al. introduced the hardware based solution, called *spin detection buffer(SDB)* for detecting meaningless spin of vCPUs produced by LHP [14]. 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 [5]. 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.

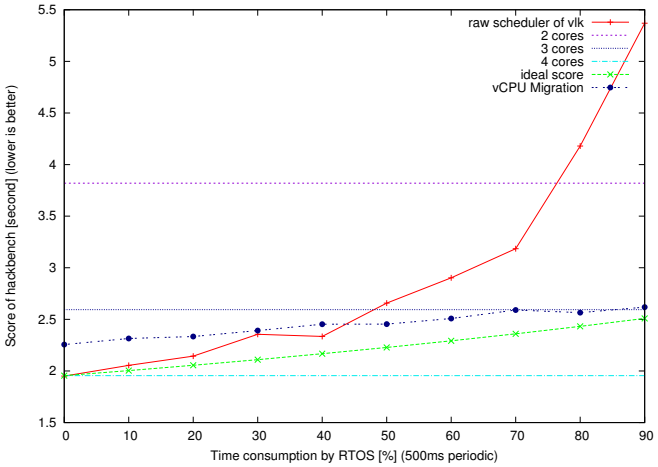

Figure 8. Result of hackbench on Various Configuration

4.2.2 Solving LHP in *vlk*

The methods described in Section 4.2.1 improve the throughput of SMP Linux on traditional VMMs. But all of them assume that there is 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 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 ordinal OSes, but in the case of *vlk*, 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 8 describes the score of hackbench on various configuration of *vlk*. In this evaluation environment, four pCPUs execute five vCPUs. Therefore two vCPU shares one pCPU. One vCPU belongs to TOPPERS and four vCPUs belong to Linux. TOPPERS executes the 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: $4-f$

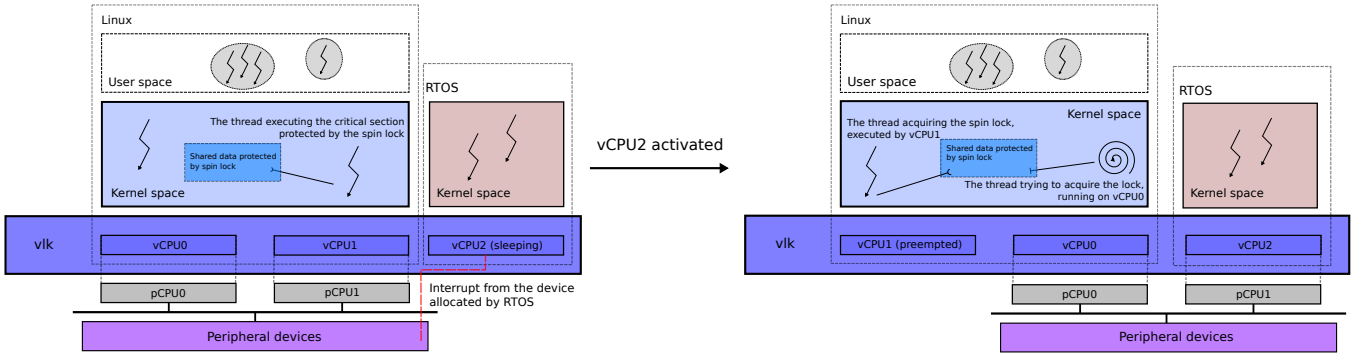

Figure 7. Typical example of Lock Holder Preemption in *vlk*

(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 that 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 *vlk*". 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 system, mutex based on busy wait mechanism is only used in kernel space. Therefore if the vCPU of Linux which shares pCPU with the vCPU of RTOS is migrated to another pCPU when the thread running on it 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 *vlk*, 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 *vlk*. In the original design strategy of *vlk* mentioned above, we gave a high priority to the vCPU of RTOS which is higher than the priority of the vCPU of Linux. when they are sharing the same physical core. But this is not always the case for that there might exist some real-time processes in Linux that have quicker response time requirements than that of the processes of RTOS. Graphical applications such as multimedia players are the most obvious examples that may require better real-time response and hard to be handled in the RTOS. In this situation, we can mark one of the vCPUs of Linux as rt-vCPU and schedule it against vCPU of RTOS. *vlk* has no clue of who is the more important vCPU of Linux, so this gives *vlk* a *hint* to recognize the urgent one and let it be scheduled quicker. 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 at the same, 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. Conclusion and Future Work

vlk can execute multiple operating systems without suffering a large amount of overhead and engineering cost.

In the original project of *vlk*, named SPUMONE, we provided optional spacial protection mechanism between guest OSes with the method named *secure pager* mainly for the security purpose [10]. With the secure pager, invalid behavior of one guest OS which cause malicious memory rewriting can be detected with low runtime overhead. The overhead of the secure pager is far lower than the spacial isolation provided in typical VMMs. However, the secure pager requires one dedicated physical CPU and the CPU cannot be multiplexed by multiple vCPUs. This constraint conflicts with the purpose of maximizing CPU resource. For satisfying the purpose of CPU utilization and seeking various possibility of design space, we are planning to implement spacial isolation between guest OSes via running them as user-level tasks like traditional VMMs and microkernels. But this isolation is not the indispensable feature of VMMs for embedded systems. Because in the case of the embedded systems, the essential resource which has to be isolated between guest OSes is not memory but mutual exclusion as we described in Section 2. Thus the spacial isolation provided by *vlk* will be configurable feature like the isolator module of VLX [3].

Another future work is also planned for the enterprise areas. During the development of *vlk*, we noticed that our approach, separating kernel space context and user space context into two physical CPUs, is employed by other researches for improving the utilization of processor resources like cache and TLBs [20, 21]. Especially the approach by Chakraborty et, al. is similar to ours. However, their implementation and measurement is done on simulator and they ignore the cost of TLB miss hit. We believe that porting our *vlk* to the real machines used in the enterprise areas and measuring the efficiency of the approach will result interesting knowledge.

References

[1] Hitoshi Mitake, Tsung-Han Lin, Hiromasa Shimada, Yuki Kinebuchi, Ning Li, and Tatsuo Nakajima. Towards co-existing of Linux and real-time OSes. In Proceedings of Ottawa Linux Symposium, 2011.

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

[3] RedBend software. Mobile Virtualization: How it Works. http://www.redbend.com/index.php?option=com_content&id=133.

[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] 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

[6] 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.

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

[8] 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.

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

[10] 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.

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

[12] 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.

[13] 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*

[14] 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

[15] 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

[16] Ingo Molnar. RT-patch. Index of /mingo/realtime-preempt:http://www.kernel.org/pub/linux/kernel/projects/rt/.

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

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

[19] Victor Yodaiken. Against Priority Inheritance. FSMLabs Technical Report. http://www.yodaiken.com/papers/inherit.pdf. 2004.

[20] Livio Soares and Michael Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10), 2010.

[21] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. Computation spreading: employing hardware migration to specialize CMP cores on-the-fly. In Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII), 2006.