Software and Hardware Supports for Multi-OS Environment

マルチOS環境を支援するソフトウェア
およびハードウェア機能の提案

February 2012

Waseda University
Graduate School of Fundamental Science and Engineering
Major in Computer Science and Engineering
Research on Distributed Systems

Yuki KINEBUCHI

Acknowledgements

This is a great opportunity to express my respect to my thesis advisor, Prof. Tatsuo Nakajima. My research and this dissertation would not exist without his support and patience. I would also like to thank all the members at Distributed Computing Laboratory in Waseda University for encouraging me.

Abstract

Personal mobile devices are rapidly enhancing their functionalities. Not limited to phone and text messages, they offer web browsing via the Internet, free to install applications on the users' requests, play musics and videos, etc. Now they are capable of running multiple OS instances with support of virtualization. Like the use in desktop/enterprise systems, virtualization for embedded mobile devices allows consolidating multiple OS instances and enhancing the security of hosted OS environment. In addition, there are applications specific to embedded systems such as hosting real-time OS (RTOS) and application OS (GPOS) concurrently without spoiling the real-time responsiveness of the RTOS.

There is no doubt that virtualization brings many benefits to the embedded mobile devices, however virtualization is not a panacea. Additional layer of virtualization incurs additional complexity to the software stack of devices. Some extra engineering efforts of developing such device might make the system prone to bugs and security risks. In this dissertation we take the position that some applications of embedded virtualization can be supported with more light-weight methods. The methods leverages architecture specific features that are common or expected to be common among embedded mobile devices. We first focus on real-time and application OS consolidation, for which we propose a thin abstraction layer that achieves better interrupt responsiveness than virtualization. Next we focus on hosting a kernel integrity monitor for rootkit detection. The monitor is hosted within an isolated memory region that is protected by means of processor architecture.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Personal mobile devices are rapidly enhancing their functionalities. Not limited to phone and text messages, they offer web browsing via the Internet, free to install applications on the users' requests, play musics and videos, etc. At this point writing this thesis the latest device is equipped with multiple processor cores run with over 1.4GHz frequency and a gigabyte of memory [6]. This outstrips the performance of desktop computers those were available in 1990s.

As their hardware advances, now they are capable of running multiple OS instances with support of virtualization [9, 17, 32]. Like the use in desktop/enterprise systems, virtualization for embedded mobile devices allows consolidating multiple OS instances and enhancing the security of hosted OS environment. In addition, there are applications specific to embedded systems such as hosting real-time OS (RTOS) and application OS (GPOS) concurrently without spoiling the real-time responsiveness of the RTOS. These benefits of virtualization motivated the ARM architecture to add a hardware virtualization support to their upcoming instruction set architecture (ISA) [62].

1.1 Motivation

There is no doubt that virtualization brings many benefits to the embedded mobile devices, however virtualization is not a panacea. Additional layer of virtualization incurs additional complexity to the software stack of devices. The hypervisor should be designed carefully to achieve reasonable performance. The straightforward port of an existing hypervisor from the desktop/enterprise field to embedded field does not perform well [34]. Extra engineering efforts of developing such device might make the system prone to bugs and security risks. In Chapter 2, we discuss the advantages and disadvantage of using hypervisors for embedded systems to accomodate multiple OS kernels in detail.

In this dissertation we take the position that some applications of embedded virtualization can be supported with more light-weight methods. The hybrid architecture [24, 66, 44, 59] and the logical partitioning of OS kernels [54] suggest the feasibility of

running multiple-OS system without support of hypervisors. Traditional pure-hypervisor based virtualization requires explicit splitting of the privilege level of a hypervisor and guest OSes. Instead we discuss methodologies for accommodating multiple-OS instances without help of a hypervisor layer. Limiting our goal to support real-time scheduling in multi-OS architecture and guaranteeing safe execution of security monitor, we leverage virtualization technologies and some architectural support to achieve our goal.

1.2 Contributions

There are two contributions in this dissertation:

- We first focus on real-time and application OS consolidation, running a real-time OS (RTOS) and an application OS (or GPOS: general-purpose OS) concurrently on a single mobile device. For which we propose a thin abstraction layer SPUMONE that achieves better interrupt responsiveness than virtualization. SPUMONE tries to minimize the modifications to the guest kernels alongside the implementation size of the virtualization layer itself. We also leveraged the architectural feature of the experimental platform to mitigate the affect of the GPOS's activities to the real-time responsiveness of the RTOS.

- Next we focus on hosting a kernel integrity monitor for rootkit detection. The monitor is hosted within an isolated memory region that is protected by means of processor architecture. We propose the limited local-memory (LLM) architecture that guarantees safe execution under a privileged processor core without support of a hypervisor layer.

1.3 Organization of the Dissertation

Figure 1.1 illustrates the structure of this dissertation. In the next chapter we discuss the tradeoffs of using virtualization in embedded systems. In Chapter 3, we first show our experience on running RTOS and GPOS on a embedded system VM that shows the overhead put into the interrupt handling. Chapter 4 follows the previous chapter, which introduces our light-weight virtualization technology that tries to eliminate the delay that we found in the experiment. The proposed methods minimize the interrupt delay caused by the interference of the application OS. In Chapter 5 we propose a method to run a security monitor without support of a hypervisor which requires a slight modification the architecture. Finally Chapter 6 concludes the dissertation.

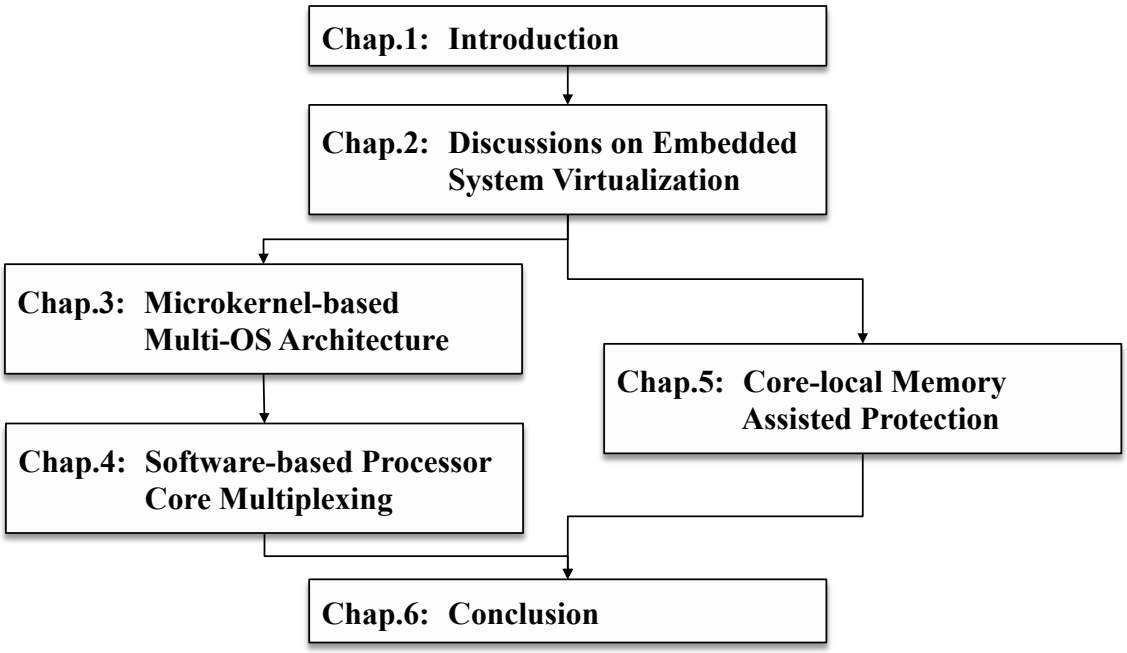


Figure 1.1: Organization of the Dissertation

Chapter 2

Discussions on Embedded System Virtualization

In this chapter, we introduce the advantages and disadvantages of using a hypervisor in embedded systems, in order to backup the motivation of our proposal on running multiple OS without help of a hypervisor.

2.1 The Use of Embedded System Virtualization

Mobile devices are capable of running multiple OS instances hosted by a hypervisor. The applications of hypervisors in the embedded systems are different from those in the desktop/enterprise world. The following describes the two major applications.

- **Real-time scheduling.** A hypervisor is leveraged for consolidating an application OS and a real-time control OS. A modern system-on-chip (SoC) for mobile devices equips application and baseband processors [7]. A baseband processor is in charge of handling mobile telecommunication signals, which usually runs a real-time OS that can handle interrupts within a hundred microsecond or even faster. Consolidating control and applications OSes to run concurrently on a single processor, the silicon vendor has an opportunity to simplify the design of the SoC. For this application, the hypervisor should be designed carefully to deliver interrupts with a reasonably small delay and to preserve the execution time of the tasks running on the RTOS. Type I hypervisor [51] is used to support this application. OKL4 [4] and virtualLogix VLX [9] are designed as type I hypervisor.

- **Security enhancement.** Virtualization may enhance the mobile device security [31]. OS can contain malicious attacks within the isolated virtual machine, not to propagate it to the remainder of the system, even when if the attacker promotes itself to the privileged mode. Since mobile devices are connected to the internet, as

the desktop/enterprise computers, we need to protect them from security attacks. Especially rootkits that stays in the system for long period of time can collect private informations.

Note that for the use in embedded systems, hypervisors need to support only a limited and fixed number of guests. For the enterprise use, it is worth running a number of application OS instances, however it is not always useful for personal users in embedded systems. A personal never use a second and a third Android unless paranoid about one's privacy, or using it in business. Some vendors use their hypervisors to integrate personal and business use cell-phones into a single device [17, 8]. Enterprises lend business phones to their employees, because personal phones may risk their security and confidentiality. Virtualization allows to install a virtual instance of a business phone into the personal phone, with which the second physical device would be unnecessary. An example of hosting two instances of Android OS is shown in [17]. Even in this case, they host guests up to two instances. Giving freedom of forking a number of virtual machines requires multiplexing the underlying peripherals and supporting virtual peripherals, which is difficult engineering because the hardware of the mobile devices varies from product to product. They use customized SoC that equips numbers of non-standardized controllers.

We take the position that an embedded system hypervisor does not require the rich functionalities we can find in the desktop/enterprise systems. Focusing our applications to the real-time support and security enhancement it is not necessary to host a complete symmetric OSes on an embedded device. Our proposal in Chapter 4 and Chapter 5 limits the use of OSes. RTOS accesses devices those are not used by the application OS, therefore it is unnecessary to virtualize peripherals. For the security purpose, the security monitor should run independently from the target OS in order to protect it from compromised by rootkits and to detect data inconsistencies. Eliminating some surplus features may contribute to simplify the design and the implementation of achieving the requirements, thus we claim these constraints and assumptions are reasonable for our background and challenges.

2.2 Performance and Real-time Responsiveness

As discussed in [14] one of the primary requirements of hardware virtualization is low overhead. This is common among embedded and desktop/enterprise environment. Since embedded systems provide less computation power than desktop/enterprise systems, the overhead of virtualization is more critical. One of the main sources of the overhead of virtualizing hardware is the trapping between a hypervisor and guest kernels for instruction emulations. Another cause of the virtualization overhead is spatial isolation among guest kernels. Switching the execution of virtual machines entails extra cache flushes and TLB misses.

In addition to low overhead, embedded systems also require real-time responsiveness. A hypervisor like Xen [16] schedules its virtual machines with a time-sharing based manner which harm the responsiveness of the guest OS. In addition, co-existing OSes may interfere the behavior of each other which may incur deadline misses. It is crucial to minimize the overhead of a hypervisor but also to minimize the affect of scheduling from the application OS to the real-time OS. In Chapter 3 we measured the real-time delay of a RTOS running on top of a hypervisor concurrently with a GPOS.

Hartig et al. [30] constructed a microkernel based system consisting of a real-time server and Linux running on top of L4 microkernel and investigated how to contain Linux's activity not to affect the real-time server. They modified Linux not to block the interrupts of processor and leveraged cache coloring to force exclusive usage of cache memory. In Chapter 4 we propose our method of reducing the real-time delay of a virtualized RTOS.

2.3 Engineering Cost

Because of the absence of the hardware virtualization support by embedded processors, a commodity embedded system hypervisor adopts para-virtualization technique. The ARMv7 architecture has a number of instructions that do not trap in the user-mode but their behaviors rely on the privileged state of the processor. These instructions are called *sensitive* instructions [52]. In the terminology introduced by the early research on the virtualization by Popek and Goldberg [51], ARMv7 is not a virtualizable architecture. These sensitive instructions can be virtualized by dynamic translation [57, 19], replace with the help of compilers [41] and rewriting them by hand which is known as para-virtualization [65, 16]. Para-virtualization requires the modification of guest OS kernels, which entails engineering cost of replacing sensitive instructions with emulating instructions and trap into the hypervisor.

A number of enterprise servers and desktop computers may share the same para-virtualized OS. This can be also benefit in embedded systems which use the same GPOS, but the RTOS varies from manufacturers to manufacturers and products to products. Considering various combinations of RTOSes and GPOSes, even though the engineering cost of constructing a single hybrid system is claimed to be small enough, the engineering cost for supporting various combinations of RTOSes and GPOSes would still introduce a great engineering effort.

Constructing Multi-OS environment needs to balance the engineering cost and the overhead. An ideal hypervisor may not require modifications to guest OSes [31]. The drawbacks of engineering cost in the para-virtulization is not found in full-virtualization. Full virtualization exposes an interface identically to underlying hardware. However it introduces large overhead to guest OSes. In addition, it requires vast modifications to the ISA of processor cores, which might be available only for some high-end SoC products

that consumes more processor power and dominates larger area on the chip.

Hybrid kernels reduce these overheads by putting OS kernels in the same privileged mode. They run a GPOS kernel as a task on a RTOS kernel. The underlying RTOS can handle interrupts in real-time while running the GPOS as a non-real-time task of it. The drawback of the hybrid approach is the modification required to the guest kernel running on top of the RTOS. Since the RTOS exposes the binary interface different from the one of the underlying processor, the architectural part should be replaced by the API of the RTOS.

This motivated us to develop a light-weight abstraction layer that leverage architectural features that requires simple extensions to the processor and some features expected to be ubiquitous among future embedded processors. We also try to minimize the modification to hosted OS kernels, which allows adopting various combinations of RTOSes and GPOSes with reasonable engineering effort. We introduce the design of our virtualization layer SPUMONE in Chapter 4 which minimizes the required modifications to the guest OS kernels.

2.4 Security

Strong isolation among OSes is an attractive feature for constructing a secure and reliable embedded system[31]. One of the approaches is to offer the spatial isolation in embedded systems is to use the microkernel. However, supporting spatial isolation with reasonable overhead requires a large amount of modification to the OS kernels. Also, for making IPC predictable, microkernels need to integrate IPC and synchronization, and it makes the microkernel complex and IPC slow as described in [47].

Despite the wide adoption as a research platform of security researches, a hypervisor itself is also the target of security attacks. As their functionality extends, hypervisor have increased their code size; now they are prone to vulnerabilities. We can find vulnerability reports on Xen and VMware at National Vulnerability Database [3]. Some vulnerabilities report the possibility of malware subverting the hypervisor layer, which means capable of gaining the complete control over the system.

In order to enhance the security of a virtualized system, it is crucial to minimize its attack surface by reducing the size and complexity of the hypervisor layer. The microkernel design can simplify the design of the hypervisor layer [32, 56]. Microkernels expose a well-defined programming interface, that are high-level abstraction of physical resources. Processors are virtualized as threads and tasks, memory management is virtualized as map/unmap function, interrupts are virtualized as IPCs, etc. These high-level API requires para-virtualization of guest kernels which is larger engineering effort than replacing sensitive instructions. Furthermore, a simple design enables verification of the microkernel. seL4 adopted formal verification to a microkernel [38]. It accomplished developing

a bug free hypervisor, however it is applicable under strict limitations. For instance, in seL4's implementation, interrupts are replaced with polling on a signal, which introduce unpredictable delay into the interrupt delivery. This limitation disallows hosting real-time tasks on the microkernel.

In Chapter 5 we work on a method to protect a security module that runs beside the target without the support of a hypervisor.

Chapter 3

Microkernel-based Multi-OS Architecture

The emergence of functional embedded systems such as cell-phones and digital appliances brought up a new issue, building a system that supports both real-time and rich services. One of the solutions is leveraging a hypervisor to integrate a real-time operating system (RTOS) and an application operating system (or GPOS: general-purpose operating system) into a single device. In this chapter we report our experience on developing a preliminary setup of such solution with a real-world machine. We reveal sources of the overheads in the pure-hypervisor based multi-OS environment in order to discuss the design of multi-OS environment suitable for embedded system, which we introduce in the next chapter. We constructed a prototype system with an existing hypervisor, an RTOS, and a GPOS, measured some basic overheads. The experiments show that the GPOS's activities entail non-negligible overhead to the delay of interrupts sent to the RTOS.

3.1 Background

In recent years, as seen in cell-phones and digital appliances, the scale and the complexity of softwares for embedded systems are increasing rapidly. These devices integrate large and complex softwares supporting functions such as network, multimedia and GUI. Despite the expansion of the software scale, software development time cycle is shortened, which entails bugs and insufficient reliability to their products. Developers are trying to leverage platforms and middleware to increase software reusability and also to extend an embedded OSes to support memory protection mechanism to increase their reliabilities. However, still the cost of their software engineering and testing is high.

To overcome these problems, GPOSes, originally targeting desktop/enterprise systems, are ported to embedded systems and already widely used. A typical example is Linux[61]. By using GPOSes in embedded systems, a wide variety of applications, net-

work protocols, libraries and middlewares developed in desktop/enterprise systems can be reused. Also they provide a memory protection mechanism which can isolate applications to increase the system reliability.

While there are various advantages leveraging GPOSes in embedded systems, some technical challenges remain. A small memory footprint, short bootup time, and especially a real-time scheduling is one of the most challenging issue. There are still many efforts being made to shorten their response time. For instance, Molnar developed the real-time preemption patch [45] for Linux, which reduces the response time of the Linux kernel by making it preemptible. According to the analysis of Abeni et al.[13], the maximum kernel latency would be 28 *milliseconds* (ms) in traditional Linux, and 17 ms in Linux with the preemption patch. Even though applying the patch to the kernel, Linux still cannot achieve a few microseconds latency which is generally supported by embedded RTOSes used in traditional embedded systems.

Cell-phones balance real-time responsiveness and rich functionality by using some additional processors. A RTOS and a GPOS run simultaneously on their own dedicated set of a processor and memory. Modern cell-phones integrate these processors and memory into a single system-on-chip (SoC) [7]. However, an additional processor increases the price of the product by dominating some area on the chip. Even if the price of a single processor is a few dollars, cell-phones are sold in order of hundred thousands, so the resulting total production cost cannot be negligible.

In recent years, leveraging hypervisor in embedded systems has been focused to overcome these problems. A hypervisor enables integrating RTOS with low latency and GPOSes with rich services by running both of them simultaneously in a single device. Some early researches on consolidating a RTOS and a GPOS on a single embedded device are done by the ERTOS[1] group at NICTA[2], developed Iguana[1], an L4 microkernel-based embedded real-time platform, and Wombat[40], para-virtualized Linux which runs in Iguana. Oikawa et al. also ported a RTOS to the L4 microkernel and to their own hypervisor. They evaluated the overhead of interrupt handling[49, 48]. The result shows that the overhead can be kept small enough. By giving a higher fixed priority to a virtualized RTOS, the real-time tasks reside in the RTOS can preserve their short response time. The hypervisor model let multiple OSes to share a single processor, which results in reducing the number of processors implemented on a device.

In this chapter, we introduce our experience on developing a multi-OS system that hosts a RTOS and a GPOS on an embedded system hypervisor. We used the above research contributions, the L4 microkernel, Wombat, Iguana. As a RTOS, we ported the work of para-virtualizing TOPPERS by Oikawa et al. that runs on the older version of the L4 microkernel to the latest version at the time. We evaluated the overhead of

[1]Embedded and Real-Time Operating Systems (http://www.ertos.nicta.com.au/)
[2]National ICT Australia (http://nicta.com.au/)

virtualization especially the interrupt latency on real-world hardware.

We also work on the problem that dispatching the RTOS prior to the GPOS in hypervisor-based systems limits real-time task deployment between guest operating systems. The task scheduling by a hypervisor is done in a unit of an OS, so it does not care about the priorities of the tasks running in guest OSes; therefore, all the high priority tasks should reside in the RTOS, and the remaining low priority tasks in the GPOS. There do exist applications for an RTOS which do not require high priority, and applications for a GPOS which requires high priority (Figure 3.1 (a)) such as an video player. Therefore we propose a task grain scheduling which enables a scheduling in a unit of a task by a hypervisor; even tasks are deployed in different guest OSes they can be prioritized (Figure 3.1 (b)). The proposed scheduling scheme shall increase the flexibility of the real-time task deployment in a hypervisor based multi-OS system, which leads to increasing the reusability of low priority applications for RTOSes and high priority applications for GPOSes.

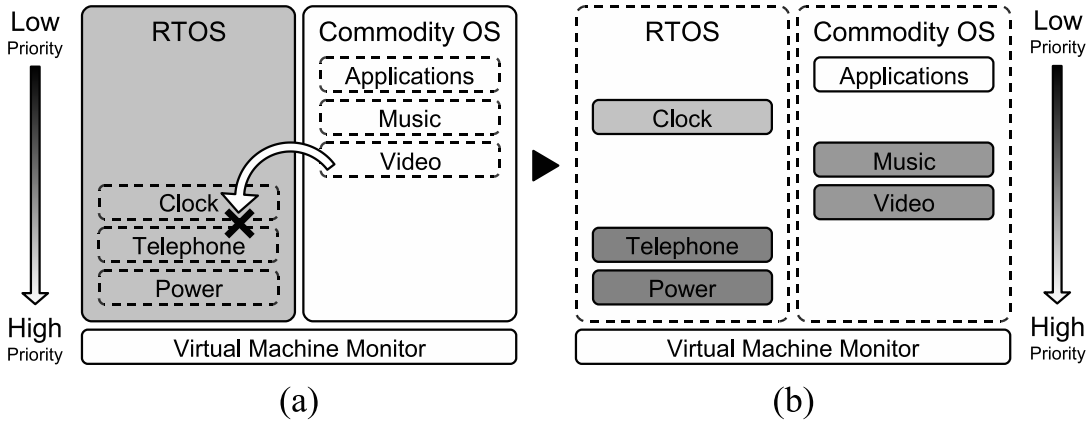


Figure 3.1: Task grain scheduling

The next section introduces some related work. Section 3.2 introduces the implementation of our prototype system. Section 3.4.3 introduces the design of the task grain scheduling. Section 3.4 introduces the results of the evaluation. Finally Section 3.6 summarizes this chapter.

3.2 Implementation

We built a prototype system with L4-embedded[2] as a hypervisor, Wombat as a guest GPOS, and TOPPERS/JSP 1.3 as a guest RTOS. In this section, we briefly introduce the hypervisor and the guest OSes, and describe how we implemented the task grain scheduling in our prototype system.

3.2.1 L4-embedded

L4-embedded (L4 in following sentences) is a microkernel which shares some common features with a hypervisor. It has a capability to run para-virtualized OSes. Wombat is one of them, a para-virtualized Linux runs on L4. L4 supports IA-32, ARM, MIPS, and PPC architectures. This time, we implemented our prototype system in the IA-32 architecture version of L4.

The main functions provided by L4 are thread management, memory management, inter process communication (IPC), and interrupt handling. Their detail follows.

- **Thread management.** A thread is a unit of scheduling done by L4. The scheduling algorithm is fixed priority preemptive scheduling. Each one of the threads has its own context. The usages of threads in virtualized guest OSes are described in Section 3.2.2 and Section 3.2.3.

- **Memory management.** L4 provides some flexible memory management functions to user space applications, creating a memory space, mapping and unmapping page frames between memory spaces, etc. Multiple threads can reside in a single memory space. Wombat leverages these functions to manage address spaces since the functions provides features equivalent to an MMU.

- **IPC.** Data passing and synchronization between threads are done by IPC. The IPC function provided by L4 can be preformed either with or without passing data. In addition, IPC can be performed without blocking (asynchronous IPC). Also software interrupts and processor exceptions are translated to IPC message by L4, and sent to corresponding threads. Systems calls and page faults triggered by Wombat processes are handled by using this mechanism.

- **Interrupt.** Hardware interrupts are passed to threads as IPC messages from pseudo IRQ threads. To which thread a specific interrupt is notified, is set by L4 API. Generally, a thread which handles an interrupt is blocking to receive an IPC message from an interrupt source. When the thread has the highest priority than all other active threads (threads in `ready` state), it is dispatched immediately receiving the interrupt message. The interrupt from the same source is masked until the thread sends back a reply message to the corresponding IRQ thread.

- **Device Servers.** When a hardware is shared by multiple applications running on L4, the access to the hardware is arbitrated by using a device server. If a thread wants to access a hardware, it sends and receives IPC messages to corresponding device server. Since an interrupt message cannot be sent to multiple threads at the same time, a shared interrupt message is first sent to the device server, and then

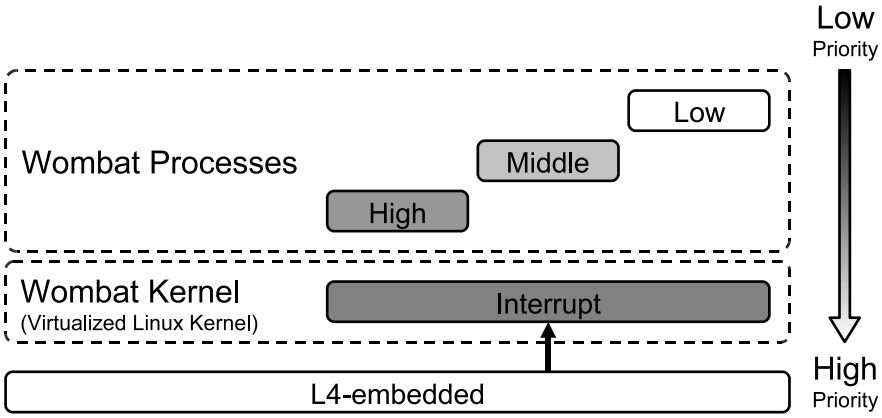


Figure 3.2: Wombat

sent to the threads. For instance, a timer interrupt is generally used by numbers of OSes. In our prototype system it is used by both Wombat and L4/TOPPERS, so the interrupt is first sent to timer server, then to one of them.

3.2.2 Wombat

Wombat is a para-virtualized Linux which runs as a server on L4. Wombat we used in the prototype is based on Linux 2.6.10. It can run unmodified Linux applications.

Wombat Threads

Wombat consists of multiple L4 threads. In this section, we describe the role of each thread.

- **Interrupt thread.** The interrupt thread handles the interrupt messages sent to Wombat. It has the highest priority among the L4 threads running Wombat. Therefore when the message is sent, the interrupt thread is dispatched immediately even if other threads are running. The default L4 priority of the interrupt thread is set to 100.

  After the L4 thread starts its execution, it calls `interrupt_loop()` and get into an infinite loop. First it blocks to wait for an interrupt message. When it receives a message, calls a corresponding interrupt handler and blocks again to wait for another message. Before it blocks, it calls `need_resched()` to check if rescheduling is necessary or not. If a task switch is going to be performed, it sends a message to the system call thread. Timer interrupt messages are sent from the timer server to the interrupt thread every 10 ms.

- **System call thread.** The system call thread handles system calls invoked by

Linux processes. Generally, system calls are implemented with software interrupt instruction. When a software interrupt is performed in a process, L4 traps and translate it to an IPC to the parent thread, which is a system call thread. When it finishes processing a system call, it sends a message back to the process and blocks again to wait for the next IPC. It is assigned L4 priority 99.

All system calls are handled by a single system call thread. To switch the context inside the Wombat kernel, the thread invokes `arch_switch()`, which switches stacks and registers without trapping into L4. A corresponding process thread will be kept in `block` state until it is dispatched again.

When the quantum of a process is expired, a rescheduling message is sent to system call thread from interrupt thread. System call thread changes the state of corresponding process thread to `stop` state and switches to another kernel context by invoking `arch_switch()`.

- **Process thread.** Process thread is a thread generated one for each Linux processes. Each Linux process has its own address space. All of them run in L4 priority 98. When it performs a system call, it block in `block` state till the reply IPC is sent. When the quantum is expired, the state of itself is changed by the system call thread to `stop` state. In this way, only one thread runs at a time.

3.2.3 L4/TOPPERS

L4/TOPPERS is a para-virtualized TOPPERS based on the porting done by Oikawa et al. In this section, we introduce the implementation of L4/TOPPERS and some extensions we made to support task grain scheduling.

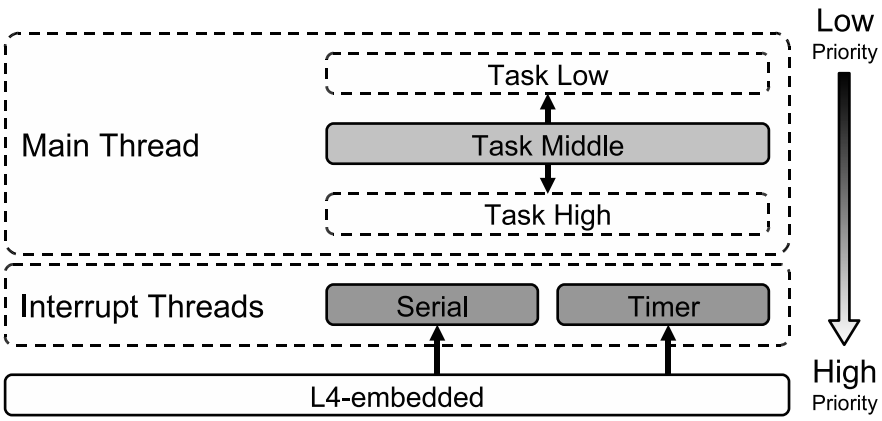


Figure 3.3: L4/TOPPERS

L4/TOPPERS consists of the main thread and some interrupt threads (Figure 3.3). All the threads run in a single address space. The main thread executes all the tasks. Here

we describe how we para-virtualized interrupt handling and interrupt locking mechanism in our system.

- **Interrupt virtualization.** Hardware interrupts are sent to interrupt threads, which they are created for interrupt source one each. All of them have a same priority. Timer interrupts are sent from the timer server.

  When an interrupt thread starts executing, it associates itself to a specific interrupt source calling L4 API, gets in an infinite loop, and blocks to wait for an interrupt message. When an interrupt message is sent to the associated thread, it disabled interrupts and invokes an original L4/TOPPERS interrupt handler. When a new thread is activated, the interrupt thread changes the `pc` register and the `sp` register. of the main thread and let it switch to a new task.

- **Interrupt lock.** Original TOPPERS executes specific processor instructions to enable and disable interrupts. Since these instructions are privileged instruction, they cannot be executed by user-level applications. Therefore in L4/TOPPERS the interrupt disabling function should be implemented as locking and unlocking a common variable shared by threads.

- **Idle state.** When there is no active thread, the main thread invokes `l4_idle()` which blocks to wait for an IPC message. Since there is no active threads in L4/TOPPERS in this state, other a thread in another OS can be executed. The main thread is resumed by sending IPC from interrupt threads.

3.3 Task Grain Scheduling

In this section, we introduce the basic design of the task grain scheduling in hypervisor. Task grain scheduling enables assigning a higher priority to a task in the GPOS running concurrently beside the RTOS on top of an embedded system hypervisor.

3.3.1 Full- and Para-virtualization

There are several different types of virtualization. One classification is full-virtualization and para-virtualization. A hypervisor supporting full-virtualization provides an interface identical to existing hardwares, so guest OSes could be run in it without any modification. The hypervisor with the task grain scheduling needs to acquire the information of guest OSes. To acquire the information in full-virtualization model, the hypervisor should know the binary layout of guest OSes and somehow trap some events such as task switches and priority changes. Since this introduce a great overhead to the hypervisor, full-virtualization is not suitable for implementing the task grain scheduling.

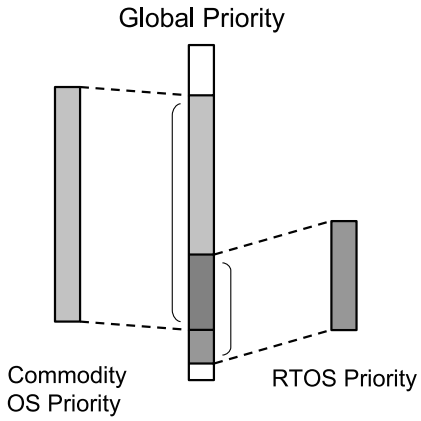


Figure 3.4: Mapping the local priorities to the global priority

In contrast, para-virtualization allows modifying guest OS source codes, so some hooks could be inserted into guest OSes to acquire information. The term para-virtualization was introduced in [65], and the basic idea already existed in early 1970s mentioned by Goldberg[26, 27]. In this chapter we consider only a hypervisor leveraging a para-virtualization technique.

3.3.2 Scheduling Algorithm

The task grained scheduling targets a hypervisor and OSes supporting fixed priority pre-emptive scheduling, since it is generally supported by major RTOSes and also GPOSes to provide real-time tasks.

3.3.3 Global Priority

Each OS has different scale of scheduling priority. Generally they are represented as integer numbers, but the value range (maximum and minimum) and the order (ascending, descending) differ. Therefore the priority of tasks running in different OS cannot be simply compared using their priority numbers. In our scheme we provide a global priority. By mapping the priority of each OS to the global priority as shown in Figure 3.4, tasks could be prioritized in a single common scale.

The priority of tasks running in different guest OS completely depends to a system configuration, so it is worthless to automate the priority mapping. The mapping should be done manually by developers during the design stage of the system. The mapping of priority numbers can be any kind of method, a simple addition or subtraction, giving a detailed mapping table, etc.

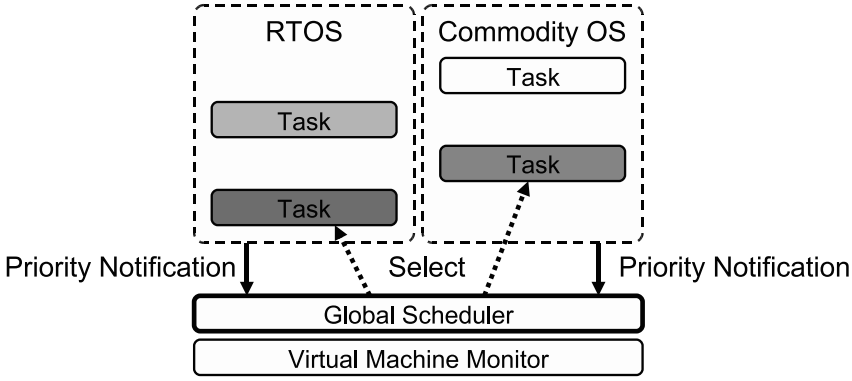


Figure 3.5: Global scheduler

3.3.4 Global Scheduler

The global scheduler, which resides in a hypervisor selects OS running a task that has the highest global priority. A guest OS tells the priority of a currently running task on it to the global scheduler when tasks switches (Figure 3.5). With the fixed priority preemptive scheduling, the task switch occurs in following timings.

- The state of a task changes from `running` to `block`
- The state of a task changes from `block` to `running`
- A task is created
- A task is deleted
- The priority of a task is changed

In other words, this mechanism is a double layered scheduling. The guest scheduling is performed as it originally does, and the global scheduler performs the scheduling when the priority change is notified by guest OSes.

3.3.5 Task Grained Scheduling in L4-embedded

In our prototype, the L4 priority is used as the global priority. The L4 priority is represented as 0 to 255 ascending (the larger is the prior) integer numbers. This range is bigger enough than the Linux and the TOPPERS priority range. Also, the L4 scheduler is used as the global scheduler. When a task is switched in a guest OS and the priority of currently running task changes, it modifies the priority of the thread by calling `L4_Set_Priority()` and the L4 scheduler is invoked by calling `L4_Yield()`.

3.3.6 Task Grained Scheduling in Wombat

Linux supports two different types of scheduling algorithms, dynamic priority scheduling and fixed priority scheduling. This time, we map Linux fixed priority to the global priority. Note that the dynamic priority scheduling is done referring to the process `nice` value which is widely used in POSIX systems, but fixed priority has nothing to do with `nice`. In default Wombat, the priority of threads running Linux processes are always 98 in the L4 priority, regardless of the fixed priority set to the process. The fixed priority of Linux is represented as 0 to 99 ascending (the larger is the prior) integer. We calculate the global priority using following formulas. $G_{W_P}(p)$ represents the global priority of a Wombat process p. $P_W(p)$ represents the Linux fixed priority of a Wombat process p. It is written as `p->rt_priority` in the Linux source code, in which `p` is a `task_struct` structure pointer for a corresponding process. C_W represents the minimum global priority that can be set by Wombat.

$$G_{W_P}(p) = P_W(p) + C_W$$

$G_{W_S}(p)$ represents the global priority of a system call thread associated with the kernel context of a process p. The global priority of the system call thread changes dynamically with which process is running in Wombat.

$$G_{W_S}(p) = P_W(p) + C_W + 1 = G_{W_P}(p) + 1$$

G_{W_I} represents the priority of the interrupt thread. It should have higher priority than the other Wombat threads. $max(x)$ represents the maximum value that x could be.

$$G_{W_I} > max(G_{W_S}(p)) = max(P_W(p)) + C_W + 1 > G_{W_P}(p)$$

3.3.7 Task Grained Scheduling in L4/TOPPERS

L4/TOPPERS notifies a priority to the global scheduler in following functions.

- `dispatch_r()` is a function invoked when a task is dispatched. A task switch involves priority change, so priority is notified to the global scheduler.
- `l4_dispatch_r()` is a function invoked when a task is resuming from preemption. It involves priority change as well as `dispatch_r()`, so the priority of resuming thread is notified to the global scheduler.
- `check_resched()` is a function invoked by an interrupt handler to check if there is a pending task switch. If a new task is going to be activated, the priority of the new task is notified to the global scheduler.

Table 3.1: Evaluation settings

	Wombat	L4/TOPPERS
Min/max priority	$C_W = 93$	$C_T = 110$
Interrupt thread priority	$G_{W_I} = 100$	$G_{T_I} = 120$

The priority of L4/TOPPERS is represented as 1 to 16 descending (the smaller is the prior) integer. $T(t)$ represents the TOPPERS priority of a task t. $G_T(t)$ represents the global priority of a task t. It is written as `t->priority` in the TOPPERS source code, in which `t` is a `task_control_block` structure pointer for a corresponding task. C_T represents the maximum global priority that can be set by L4/TOPPERS.

$$G_T(t) = C_T - T(t)$$

G_{T_I} represents the priority of interrupt threads. It should be greater than the other TOPPERS threads.

$$G_{T_I} \geq C_T$$

$$C_T > max(G_T(t)) = C_T - 1$$

3.4 Evaluation

We made evaluation with the prototype system in Section 3.2. The machine we used is DELL Precision 390; Intel Core2 Duo E6600 2.4GHz CPU, 2GB Memory and ATA133 512GB HDD. Core2 Duo is a dual core processor, but we used only one of the cores. To acquire the time we leveraged the `rdtsc` instruction which gives a time stamp which has an resolution of CPU ticks. We divide it with the frequency of the CPU and show them in *microseconds* (μs) or *milliseconds* (ms).

In the evaluation, C_W (the minimum global priority of the Wombat tasks), G_{W_I} (the global priority of the Wombat interrupt thread), C_T (the maximum global priority of the L4/TOPPERS tasks) and G_{T_I} (the global priority of the L4/TOPPERS interrupt threads) are set as shown in Table 3.1. C_W is 93 so it may expire G_{W_I}, but since the prototype system use only the Linux fixed priority from 0 to 5, it does not corrupt the scheduling.

Section 3.4.1 discusses about the context switch overheads. We evaluate direct and indirect interrupt delay as the basic overheads in Section 3.4.2. In Section 3.4.3, we run a dummy task set to try prioritizing tasks running in two different guest OSes. Section 3.4.4 evaluates the effect of frequent disk access performed in Wombat to a cyclic task running

in L4/TOPPERS.

3.4.1 Context Switch Overhead

In case of L4/TOPPERS without the task grain scheduling, the basic context switch overhead is equivalent to original native TOPPERS. However since interrupts are interposed by L4, the task switch triggered by an interrupt would take longer time than native TOPPERS. The overhead of the interrupt handling is shown in Section 3.4.2. Also, the context switch overhead of Wombat is described in [40] in detail.

When the task grain scheduling is enabled, system calls are invoked at every task switch to notify the priority of running task to the L4 scheduler. Therefore the time of an L4 system call is added to task switch overheads, for both L4/TOPPERS and Wombat.

3.4.2 Interrupt Delay Overhead

The interrupt handling delay is increased when an OS is virtualized, because the interrupt is interposed by the underlying hypervisor. In this section, we measured and compared the delay of invoking interrupt handler for original TOPPERS (Figure 3.6 (a)) and L4/TOPPERS (Figure 3.6 (b)). We set measuring points at followings.

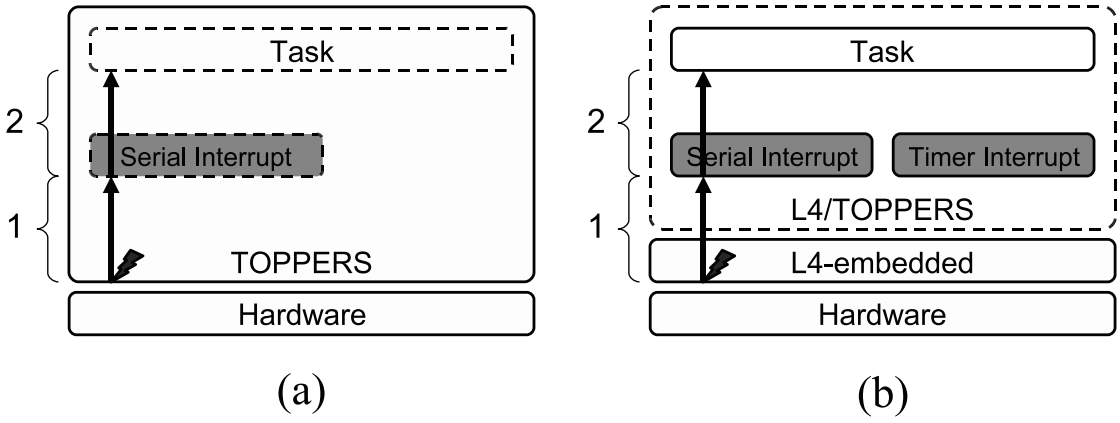


Figure 3.6: Serial interrupt delays

- TOPPERS
  - The entry point of serial interrupt
  - The beginning of serial handler
  - The beginning of a task waiting for serial input
- L4/TOPPERS
  - The entry point of serial interrupt in L4

– The beginning of the serial handler

– The beginning of a task waiting for serial input

The average of the delays are shown in Table 3.3. The overhead is 3.46μs. The processing time between the entry point and the handler increases because of the virtualization. The increased overhead between the handler and the application is because the interrupt enabling and disabling instructions are replaced by alternative functions.

Table 3.2: Average interrupt delay: in *microseconds* (*cycles*)

	TOPPERS/JSP	L4/TOPPERS
1. Entry point - Handler	0.05 (124)	2.45 (5867)
2. Handler - Task	11.54 (27623)	12.60 (30178)
Total	11.59 (27747)	15.05 (36045)

Table 3.3: Worst interrupt delay: in *microseconds* (*cycles*)

	TOPPERS/JSP	L4/TOPPERS
1. Entry point - Handler	0.07 (171)	11.39 (27270)
2. Handler - Task	19.28 (46170)	13.95 (33390)
Total	19.35 (27747)	25.34 (60660)

We measured the overhead of interrupt handling with a device server (Figure 3.7 (b)), compare it with the direct interrupt handling (Figure 3.7 (a)).

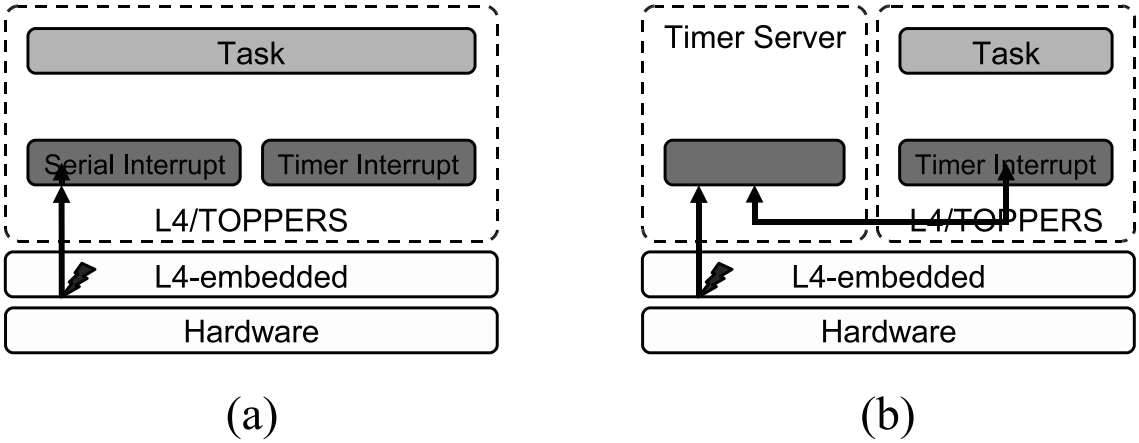


Figure 3.7: Direct and indirect interrupt delays

The delay of passing an interrupt message to an user-level thread is almost same for indirect and direct (Table 3.4). The IPC passing from timer server to TOPPERS takes

9.18μs. Therefore the overhead of passing an interrupt message through a device server is 8.98μs.

Table 3.4: Average indirect timer interrupt delay: in *microseconds*

	Direct	Indirect
Entry point - Thread	2.33	2.25
Timer server - Handler (Indirect)	-	9.18
Interrupt thread - Handler (Direct)	0.12	-
Total	2.45	11.43

The measurements show that the delay of handling interrupts directly is increased 3.45μs (8298 cycles) because of the para-virtualization, and the overhead of handling interrupt through device server is 8.98μs. Comparing these values with the original TOPPERS, the delays are increased 30% and 90%. Whether this overhead is acceptable in constructing a system, depends on the requirement of a task set in the system. Note that the overhead is rather small comparing to the jitters in the Linux kernel. If this overhead is not negligible, one way to decrease it is to use a processor which has a higher frequency. Another way is to redesign the system not to share an interrupt source, or make the OS itself a device server.

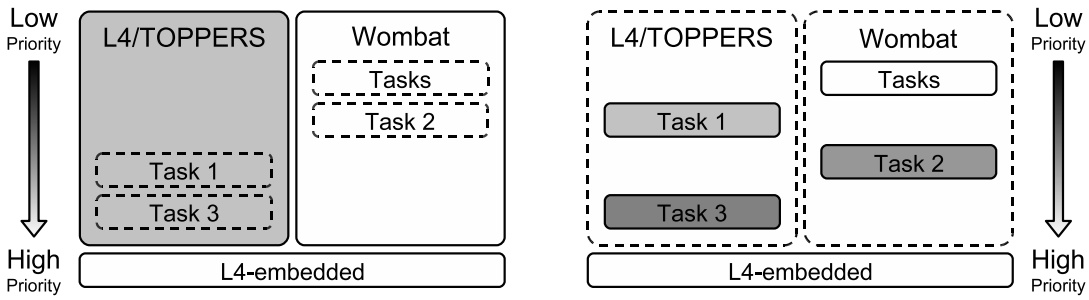
3.4.3 Task Grain Scheduling

We observed the behavior of the task grain scheduling applying it to tasks shown in Table 3.5. This evaluation is done in DELL Precision 390, the machine described in detail at the top of this Section 5. Without the task grain scheduling, scheduling by the hypervisor is done in an unit of OSes (Figure 3.8 (a)), and with it done in an unit of tasks (Figure 3.8 (b)).

Task 1 and 3 are TOPPERS tasks which is activated in cycles shown in the table. Every time it is activated, it executes an empty loop for the execution time shown in the table.

Task 2 is a real-time process running in Wombat. It executes an empty loop for 12ms.

We measured the time that tasks are activated and stopped without and with the task grained scheduling. The result is in Figure 3.9. To meet the deadline of Task 1, high priority is set to L4/TOPPERS. Therefore Task 1 gets a higher priority than Task 2, so the Task 2 is preempted when Task 1 is active. Thereby Task 2 misses its deadline as shown in Figure 3.9 (a).



(a) Without task grain scheduling (b) With task grain scheduling

Figure 3.8: Dummy tasks without and with task grain scheduling

Table 3.5: Dummy tasks

	OS	Cycle	Execution time	L4 priority (w/o TGS)	Global priority (w/ TGS)
Task 1	L4/TOPPERS	$60ms$	$20ms$	110 (High)	95 (Low)
Task 2	Wombat	$30ms$	$12ms$	98 (Low)	96 (Middle)
Task 3	L4/TOPPERS	$2ms$	$100\mu s$	110 (High)	105 (High)

Figure 3.9 (b) shows the measurement with task grain scheduling. Task 2 is assigned a higher global priority than Task 1, so Task 2 is not preempted by Task 1. In this way, all the tasks meet their deadlines.

If we give following assumptions to the task set in Table 3.5, rate monotonic scheduling[39] can be applied.

- The tasks do not have any shared resources
- Deadlines are equal to cycles
- No overhead in task switch

The sum of CPU usage is,

$$0.1/2 + 10/30 + 20/60 = 0.71666...$$

Using the formula of rate-monotonic scheduling,

$$U = 3(\sqrt[3]{2} - 1) = 0.77976...$$
$$0.71666... < 0.77976$$

So these tasks can be guaranteed not to miss their deadlines.

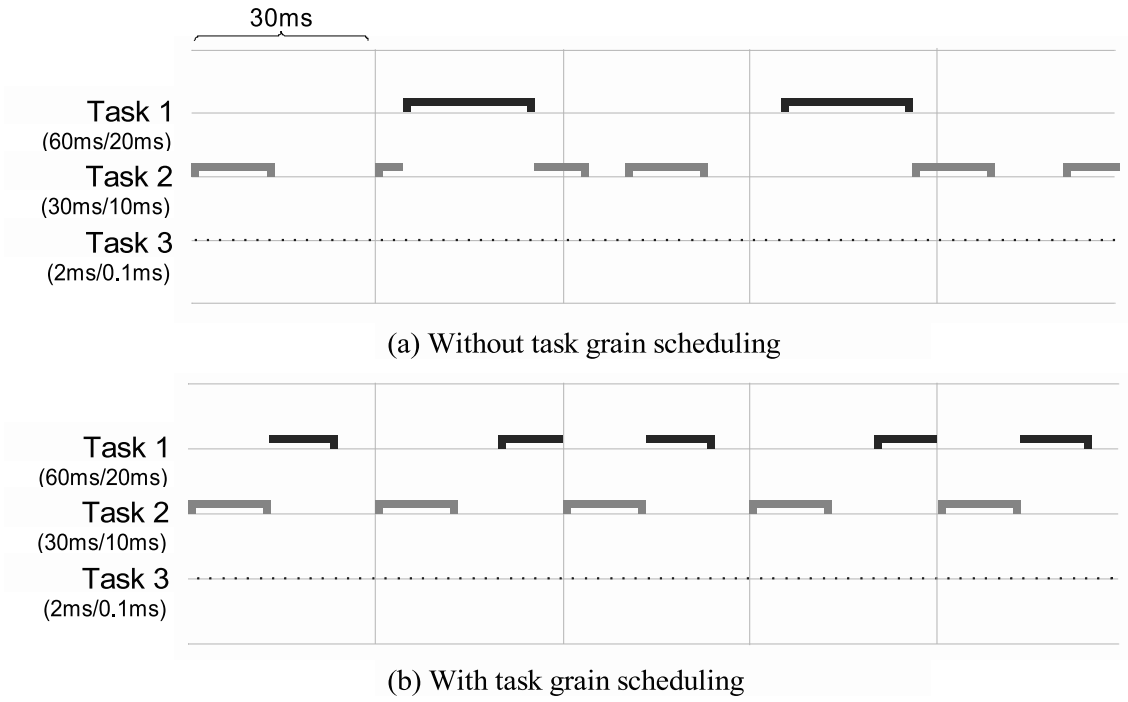


Figure 3.9: Testing task grain scheduling

Since rate-monotonic scheduling has strict constraints, they are difficult to use in practical systems. However we believe the task grain scheduling increased the flexibility of priority setting in hypervisor-based system. If they try to overcome these problem without leveraging the task grained scheduling, task should be ported to another OS or task should be split to multiple tasks which both take big engineering costs.

3.4.4 Interrupt Delay Jitters

We measured the jitters introduced to the interrupt delay of para-virtualized TOPPERS. In the experiment, we run L4/TOPPERS and Wombat concurrently to see the effect of activities within Wombat. In this setup, all the tasks on L4/TOPPERS have higher priority than those on Wombat, which does not use the task grain scheduling. Critical sections in L4 could delay the dispatch of interrupt threads. When the API invocation in Wombat and the interrupt associated to L4 occurs in a same time, the dispatch of the L4/TOPPERS thread can be delayed.

We measured the intervals of L4/TOPPERS's periodic task which runs every $2ms$, with running Wombat heavily loaded with a program accessing a HDD. We compare the cycle of the task with running them with the The y-axis shows the interval time from the previous dispatch time to the dispatch time of a cyclic task. The x-axis shows the iteration, how many times the cyclic task is dispatched. The worst case experienced $19\mu s$

(44198 cycles) additional delay over the average interval of 2017μs (4829941 cycles).

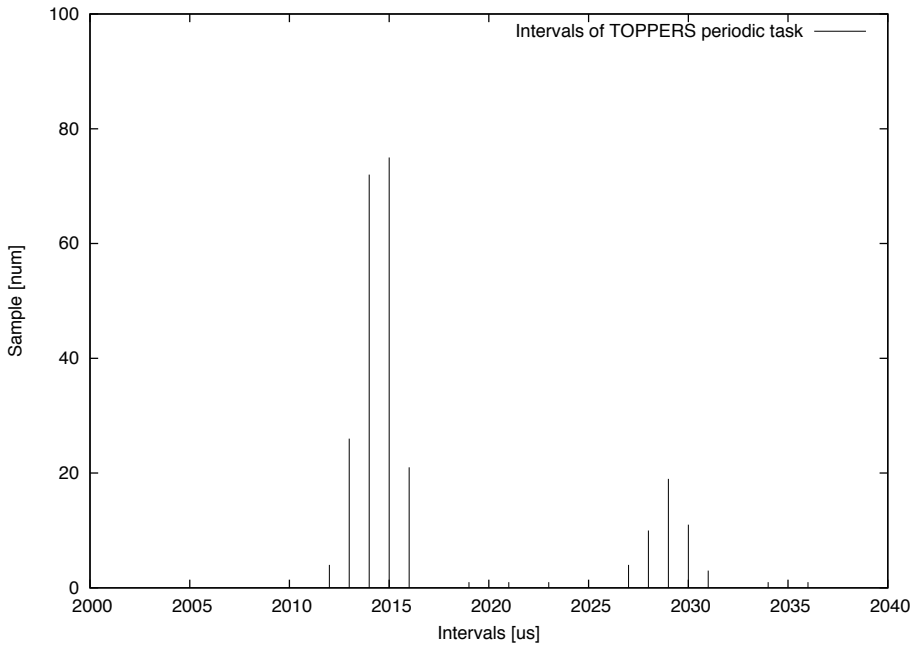


Figure 3.10: The intervals of TOPPERS's 2ms periodic task.

3.5 Related Work

In this section, we classify related work into three different groups, and compare with our approach.

3.5.1 Improving OS Real-time Performance

There are some efforts made to modify an existing GPOS to support real-time scheduling. As we mentioned in Section 3.1, Molnar developed the Linux real-time preemption patch [45] which enables Linux to preemption processes in the kernel area. Ishiwata et al. extended the Linux kernel to support hard real-time tasks by replacing spin locks with mutex locks that support the priority inheritance [35, 36]. Some commodity products are also supporting extension to satisfy short response time required by embedded systems, such as MontaVista Linux [46].

In this approach, the task scheduling could be done in a unit of tasks (or process, in POSIX systems). However supporting both real-time and high throughput are con-

tradicting concept. Supporting both of them at a time entails a complex system design, implementation and great engineering cost. Furthermore comparing to the hypervisor-based systems, applications for different OSes should be ported, or black box applications for other OSes can not be used.

3.5.2 Hybrid Architecture

RTLinux [24], RTAI [44] are real-time extended Linux implementation, which are interpolating a microkernel between the Linux kernel and the underlying hardware to achieve the short interrupt response time and guaranteed real-time scheduling. RTLinux is replacing a part of Linux's hardware abstraction layer by their own real-time microkernel. In contrast, RTAI is implemented as Linux modules and giving minimal modification to the Linux kernel, though their basic ideas are the same. In this model, Linux processes never affect the behavior of the microkernel and its tasks, since the interrupt disabling instructions are removed from the Linux kernel and it only can disable the interrupt virtually. All the interrupts are first send to the microkernel and then to the Linux kernel. In this case, the priority of the Linux processes are always the lowest.

Linux on ITRON is Linux running as a task of the ITRON real-time OS[59]. The architecture of Linux on ITRON is similar to the technique taken by RTLinux and RTAI. The major difference is that Linux on ITRON is using an existing RTOS based on μITRON specification as its interpolation layer. This gives an additional advantage of reusing existing applications compatible with μITRON specification. Some discussions about the coarse grain scheduling of real-time tasks were discussed in the paper, but they have not implemented fine grained scheduling in their system.

Comparing to the hypervisor-based approach, hybrid architecture lacks flexibility to select an RTOS. RTLinux and RTAI provides its own API to real-time processes, which are not compatible with existing APIs. Linux on ITRON provides the μITRON API to real-time processes, but no choice to use other RTOS APIs.

3.5.3 Hypervisor-based Systems

A hypervisor is a system level software which enables multiple OSes to share hardwares, such as a processor, memory, and sometimes other peripheral devices. Each guest OSes runs in an isolated address space, so the fault of an OS does not affect other OSes. This characteristic is anticipated to make embedded systems more robust.

When the hypervisor itself supports real-time scheduling, it could be potentially used as a platform for running real-time applications. This is mentioned by David Golub et al. in the paper introducing an Unix server on the Mach microkernel[28]. Also, some efforts were made to make Mach real-time[47].

Xen[16] is a hypervisor developed for servers and desktops. It supports both para-

virtualization and full-virtualization. With para-virtualization, it can run modified Linux (XenoLinux) as its guest OS. Applications can run on XenoLinux without any modifications. Both full virtualization and para-virtualization is supported. A group in Samsung has experimentally ported Xen to the embedded ARM architecture, whose purpose was to enhance the security of devices [34]. The experiment does not discuss on interrupt lantencies. The scheduling of Xen aims to sustain system throughput, but not real-time responsiveness. The main scheduling algorithm is BVT[23] which sets a weight to each OS as a parameter. The processor usage of each OS get close to the ratio of weight after running for awhile. In this chapter we target embedded systems which require real-time scheduling. Achieving guaranteed real-time scheduling and high system throughput are contradicting concept.

L4Linux is a Linux kernel implemented on top of the L4Ka microkernel as a server[29]. Some additional work has been done for integrating real-time applications and L4Linux on top of L4Ka[30]. The purpose of this research was not scheduling granularity but good performance isolation. They 'tamed' Linux not to affect real-time tasks running beside it on top of L4Ka.

Oikawa et al. ported the TOPPERS/JSP kernel [3] [60] (TOPPERS for short) to the TL4 microkernel (TL4 for short) which is a modified L4Ka::Hazelnut kernel[58]. In other words, they built a TOPPERS interface on top of L4. TL4 has a capability to run multiple instances of TOPPERS. Each of them runs in an isolated memory space provided by TL4. Therefore, the entire TOPPERS kernel and its applications can be directly reused. They measured the delay of interrupt handling and task dispatch. The results show that the overhead is small and delay is close to the original TOPPERS. This research shows that the overhead of virtualization can be reasonable value.

Iguana is a real-time embedded platform constructed on the L4-embedded microkernel. In parallel, para-virtualized Linux, called Wombat, is developed on top of Iguana. Iguana was developed to facilitate the programming in the L4-embedded based system since L4-embedded supports only primitive mechanisms. Our work is using Iguana and Wombat as its platform. We ported the work of Oikawa et al. to the Iguana platform, and let TOPPERS run on L4-embedded beside Wombat.

Today, products such as VLX [8] and VMware Mobile [17] are available. The L4-embedded project is now branched to the OKL4 project [4].

3.6 Summary

In this chapter we introduced our experience on developing a multi-OS architecture for embedded system that hosts a RTOS and a GPOS. We evaluated the real-time responsiveness of our implementation on the real-world machine. The result shows that the

[3] TOPPERS/JSP is an embedded RTOS compatible with μITRON specification

hypervisor introduced large interrupt delay overhead into the interrupt response time of the RTOS. In addition the interrupt source shared between the OSes entailed jitters into the cycles of the periodic task. This is the limitation of the architecture which requires sharing an interrupt source between guest kernels.

We also developed the task grain scheduling on our implementation, which give flexibility of assigning a priority higher than tasks on the RTOS to tasks on the GPOS. Adopting the method we succeeded to let the experimental task set to fulfill the requirement of applying the rate-monotonic scheduling to meet the deadline.

Chapter 4

Software-based Processor Core Multiplexing

Despite the strong requirement of supporting deterministic real-time scheduling on virtualization based multi-OS embedded systems, which enables co-location of a RTOS and a GPOS on a single device, there are few investigations in real-world hardware. In this chapter we introduce our virtualization layer SPUMONE, which runs on single-core and multicore SH-4A processors which introduces low overhead and requires small engineering effort to modify guest OS kernels. SPUMONE now can execute the TOPPERS RTOS and Linux as a GPOS concurrently on a single embedded test platform. In addition we propose two methodologies to mitigate the interference of Linux to the real-time responsiveness of the RTOS. One leverages the interrupt priority level mechanism supported by the SH-4A processor. The other is the proactive migration of virtual core among physical cores to prevent the Linux kernel activity from blocking the interrupts assigned to the RTOS. The evaluation shows our methodologies can decrease the interrupt delay of the RTOS entailed by Linux. In addition, sharing a core between the RTOS and Linux will increase total processor utilization when executing some specific applications.

4.1 Background

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

Therefore, constructing an embedded device with a real-time and a GPOS has attracted attention as an approach to let embedded device balance real-time responsiveness and rich functionalities. There are various approaches to achieve this. One of the approaches is to use a multi-core SoC typically equipped with two processors, one for a

RTOS and the other for a GPOS. Another approach is the hybrid system[66, 44, 59] which executes a general purpose OS as a task of a real-time OS.

In this chapter we focus on virtualization technologies, originally widely used in enterprise servers and desktop computers. Now, embedded systems are attracting attention as a new research field of virtualization technologies [31]. Embedded systems require different characteristics and gives some new challenges those have not been discussed on the previous application fields of virtualization technologies. According to the discussion in [14], the requirements to embedded system hardware virtualization are;

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

Hypervisors for enterprise servers and desktop systems, like VMware and Xen, do not fulfill these requirements. Especially the third requirement is difficult to be supported by functional hypervisors. Because virtual memory virtualization and I/O virtualization require complex manipulation of data structures inside hypervisors, they require to synchronize the data structures, and make the hypervisor complex. Therefore, we need to develop a virtualization layer specialized for embedded systems which is not based on the methodology of traditional pure-hypervisors that run in the most privileged level and isolate virtual machines.

We developed a virtualization layer on top of a real-world embedded device and evaluated its real-time responsiveness. There are three contributions introduced in this chapter.

- The first one is an OS consolidation methodology which fits the requirements of embedded systems. The evaluation shows that basic overhead and engineering required to the guest OSes are significantly small compared with related work.
- The second contribution of this chapter is an investigation on the real-time properties of virtualization technology on real-world devices.
- The third contribution is our proposal of two methodologies for decreasing the overhead introduced to the RTOS. One is a methodology that leverages interrupt priority level (IPL) mechanism to enable RTOS to preempt GPOS's critical section. The other is to migrate virtual cores among physical cores, when they enter a critical section, in order to prevent GPOS kernel activities to block the execution of RTOS.

We developed a thin virtualization layer called SPUMONE which enables the co-execution of multiple OSes on single-core processor and multi-core processor equipped

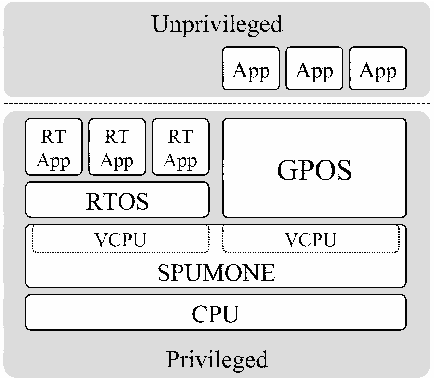


Figure 4.1: SPUMONE based system on a single-core processor

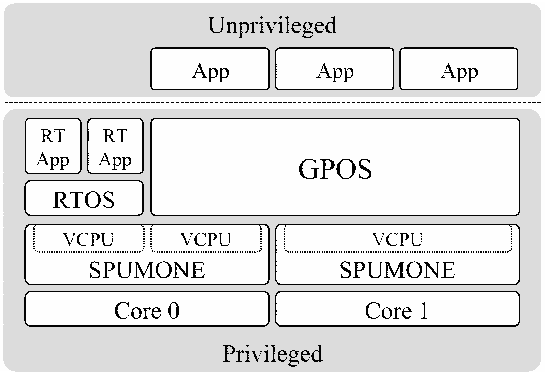


Figure 4.2: SPUMONE based system on a multi-core processor

with SH-4A architecture cores. SPUMONE can co-execute the TOPPERS RTOS [1] and Linux. The evaluation shows that our approach achieves sufficient real-time responsiveness for both methodologies. However, virtual core migration introduces a large overhead depending on the property of application executed on top of SMP Linux, which decreases the performance of GPOS applications.

4.2 Design and Implementation

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

4.2.1 SPUMONE

SPUMONE (Software Processing Unit, Multiplexing ONE into two or more) is a thin software layer for multiplexing a single physical processor into multiple virtual ones. In other words, SPUMONE provides a virtual multicore processor interface on top of a physical single-core processor. Unlike typical hypervisors or virtual machine monitors, SPUMONE itself and OS kernels are executed in privileged mode as shown in Fig.4.1, in order to simplify the system design and to eliminate the overhead of cross domain calls between user and kernel mode for system-calls and hypercalls. If an OS does not leverage privilege levels, its applications will be executed in kernel mode altogether.

Executing the virtualization layer and the kernels in the kernel mode contributes to minimize the overhead introduced to and the amount of modifications required to the OS kernels. Furthermore it makes the implementation of SPUMONE itself simple. Executing

[1] TOPPERS is a RTOS which meets μITRON RTOS specification widely used in Japanese industry http://www.ertl.jp/ITRON/

OS kernels in user mode is known to complicate the implementation of the abstraction layer, because various privileged instructions need to be emulated.

In SPUMONE based system, the majority of the kernel and application instructions, including the privileged instructions, are executed directly by the real processor, and only a minimal set of instructions are emulated by SPUMONE. These emulated instructions are invoked from the OS kernels using a simple function call. Since the interface has no binary compatibility with the original processor interface, we simply modify the source code of OS kernels, a method known as the para-virtualization Thus we assume we have access to the source code of the OS kernels. The modifications required to the OS kernels are described in details in Sec.4.2.2.

SPUMONE for multicore processors is designed in a distributed model: a dedicated instance of SPUMONE is assigned to each physical core as shown in Fig.4.2. This design is chosen in order to eliminate the unpredictable overhead of synchronization among multiple processor cores. In addition, the basic lock mechanism can be shared between single-core and multicore version, which may the simplify the design of SPUMONE. It also enables the system to scale on multicore and many-core processors as discussed in [18].

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

  To let SPUMONE receive interrupts before the OSes, we modified the interrupt entry point of the OS kernels to SPUMONE's vector table. The entry point of each OS is notified to SPUMONE via a virtual instruction for registering their vector table. An interrupt is first handled by SPUMONE's interrupt handler in which the destination virtual processor is decided and the corresponding scheduler is invoked. When the interrupt triggers an OS switch, all the registers of the current OS are saved into the register stack, then the register stack for the other OS is loaded. Finally the execution branches into the entry point of the destination OS. The processor registers are setup just as the real interrupt occurred, so the code of the OS entry points does not need to be modified.

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

- **Virtual Processor Scheduling** A processor is multiplexed by scheduling the exe-

cution of OSes. The execution states of the OSes are managed by a data structure that we call a virtual processor or a virtual core. When switching the execution of the virtual processors, all the hardware registers are stored into the corresponding virtual processor's register table, and then loaded from the table of the next executing virtual processor. The mechanism is similar to the process paradigm of a classical OS, however the virtual processor saves the entire processor state, including the privileged control registers.

The scheduling algorithm of virtual processors is a fixed priority preemptive scheduling. When the RTOS and the GPOS share a core, the virtual processor bound to the RTOS would gain a higher priority than the virtual processor bound to the GPOS in order to maintain the real-time responsiveness. This means the GPOS is executed only when the virtual processor for the RTOS is in an idle state and has no task to execute. The process or task scheduling is left up to OS so the scheduling model for each OS is maintained as-is. The idle RTOS resumes its execution when it receives an interrupt. The interrupt for RTOS preempts the GPOS immediately, even if the GPOS is disabling interrupts. When virtual cores assigned to GPOS are migrated to execute on a shared core, those cores are scheduled with timesharing scheduler.

- **Inter-core Communication** Communications among SPUMONE instances running on their physical cores are implemented with shared memory area and inter-core interrupt (ICI). First, sender stores data on specific memory area, then sends an interrupt to the destination, and receiver loads data. Or receiver may also wait for sender by polling on data.

4.2.2 Modifying OS Kernels

Each OS is modified to be aware of the existence of the other OS, because hardware resources other than the processor are not multiplexed by SPUMONE. Thus those are exclusively assigned to each OS by reconfiguring or by modifying their kernels. The following describes the points of the OS kernels to be modified in order to run on top of SPUMONE. Table 4.1 lists the modified source code of Linux.

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

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

- **Physical Memory** A fixed physical memory area is assigned to each OS. The physical address for the OSes can be simply changed by modifying the configuration

file or their source code. Virtualizing the physical memory would impose a large code into the virtualization layer and substantial performance overhead. In addition, unlike the hypervisor for enterprise systems, embedded systems have a fixed number of OSes. For these reasons we assigned fixed physical memory area for each OS.

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

- **Peripheral Devices** Peripheral devices are assigned by SPUMONE to each OS exclusively. This is done by modifying the configuration of each OS not to share the same peripherals. We assume that most of devices can be assigned exclusively to each OS. This assumption is reasonable because embedded system multi-OS platforms have asymmetric OS combinations unlike a symmetric multi-OS platform for enterprise systems. It consists of different kinds of OSes, usually a RTOS and a GPOS. For instance, a RTOS is used for controlling specific peripherals such as a radio transmitter and some digital signal processors, and a GPOS for controlling a display and buttons.

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

- **Reset Vectors**

  SPUMONE also runs on multi-core processors. The design of multi-core SPUMONE is basically the same as the single-core version. Each core is managed by a dedicated SPUMONE instance. Interrupts are handled by the instance bound to each core, then forwarded to the OS. Each instance communicates using inter-core interrupt (ICI) and shared memory area. The original processor mechanism of resetting a core is replaced with SPUMONE's function.

Table 4.1: A list of the modifications to the Linux kernel

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

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

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

4.3 Interrupt Delay Reduction

4.3.1 Interrupt Priority Level Assignment

In order to contain interrupt delay of a RTOS in reasonable value independent of the activity of a GPOS running concurrently on a single device, we propose two methodologies. First is replacing the interrupt enable and disable instructions with the virtual instruction interface. A typical OS disables all interrupt sources when disabling interrupts for atomic execution. By contrast, our approach leverages the interrupt mechanism of the processor: we assign the higher half of the interrupt priority levels (IPLs) to the RTOS and the lower half to the GPOS (Fig.4.4). When the GPOS tries to block the interrupts, it modifies its interrupt mask to the middle priority. The RTOS may therefore preempt the GPOS even if it is disabling the interrupts (Fig.4.3 (1)). On the other hand when the RTOS is running, the interrupts are blocked by the processor (Fig.4.3 (2)). These blocked interrupts could be sent immediately when the GPOS is dispatched.

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

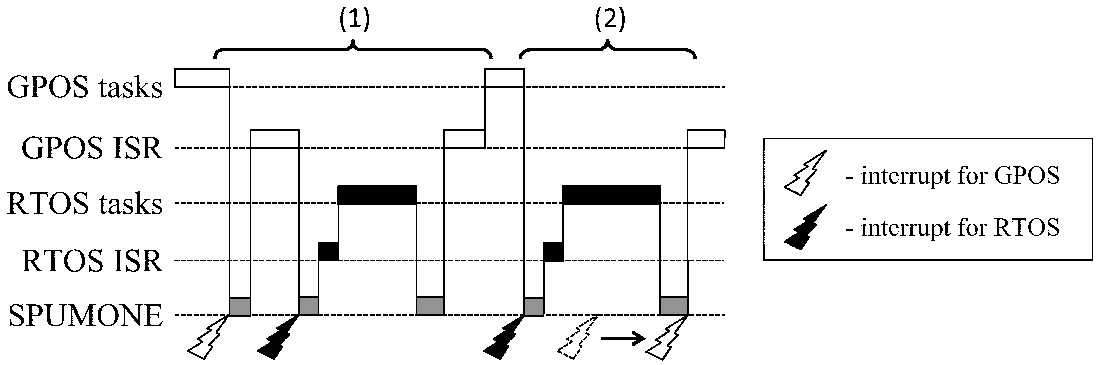


Figure 4.3: Interrupt Delivery Mechanism

execution. Figure 4.4 shows the interrupt priority levels assignment for each OS, which we used in the evaluation environment.

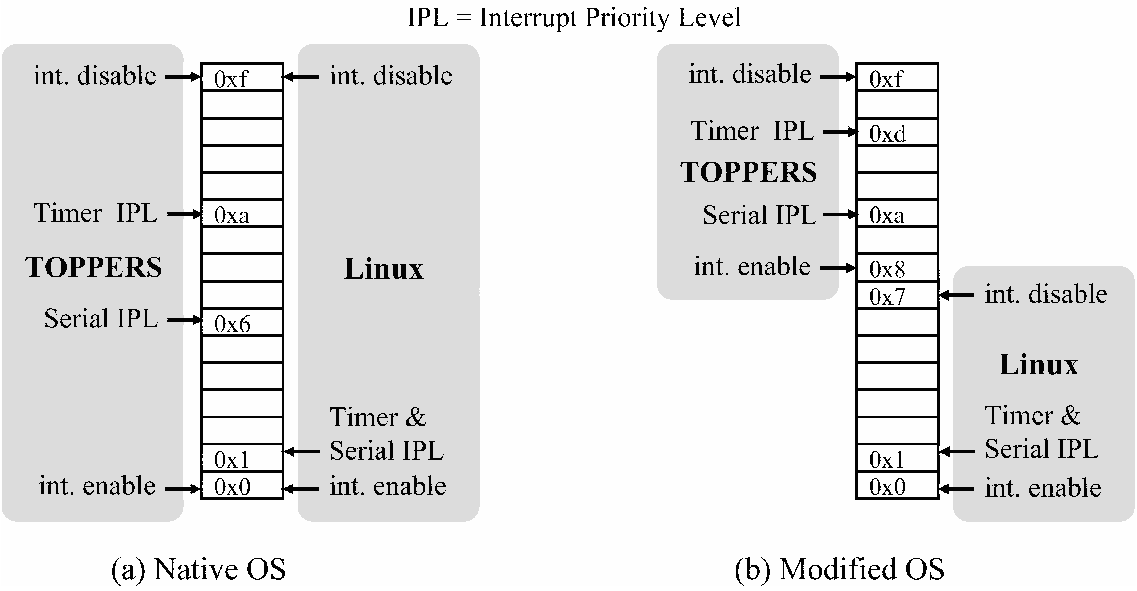


Figure 4.4: The interrupt priority levels assignment

4.3.2 Virtual Processor Core Migration

Second methodology is based on virtual core migration. As we implemented the IPL methodology, we found out that some paths of the GPOS kernel gained a highest lock priority unexpectedly (e.g. bootstrap, idle thread). This suggests us the possibility that some device drivers or kernel modules programmed in bad manner gains a high IPL and interfere with the activity of the RTOS. We modified SPUMONE to proactively migrate virtual core, which is assigned to GPOS sharing a physical core with RTOS, to another

core when it traps into the kernel or interrupts are triggered. In this way, only the user level code of the GPOS is executed concurrently on a shared core, which will never obtain a lock. Therefore, the RTOS may preempt the GPOS immediately without separating IPLs (Fig.4.5).

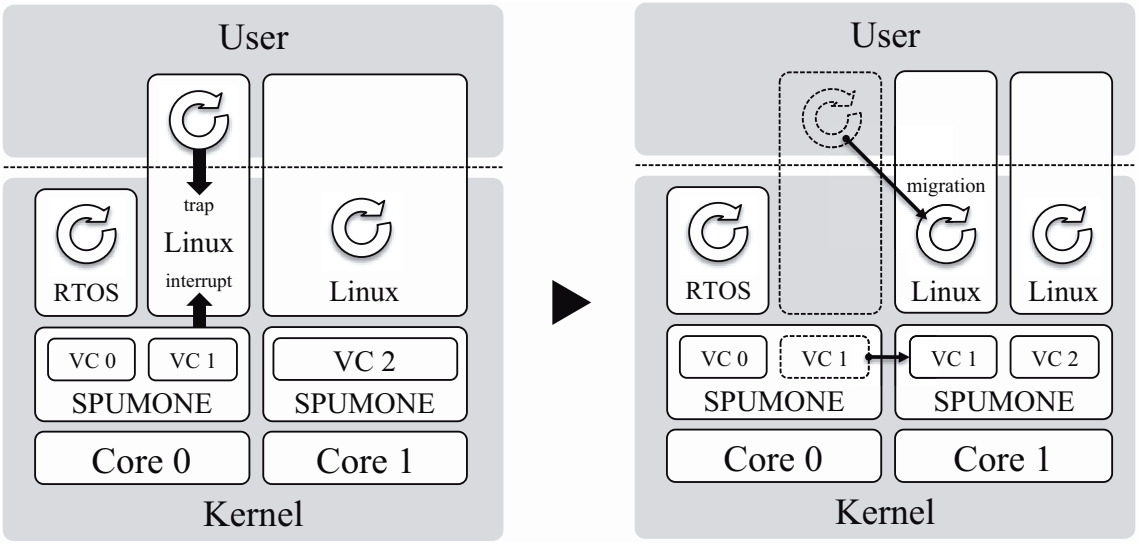


Figure 4.5: Virtual core migration

4.4 Evaluation

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

4.4.1 Basic Overhead

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

Table 4.2 shows the average and the worst case CPU cycles required to handle the interrupts sent to native TOPPERS and modified TOPPERS. In the average case

SPUMONE imposes 0.67μs overhead to the delay. The worst case overhead shows the time required to save the state of Linux and restore the state of TOPPERS. The increased delay is sufficiently small and predictable for executing real-time applications.

Table 4.2: The delay of handling the timer interrupts in TOPPERS.

Configuration		CPU Clocks	Time (μs)
native TOPPERS (SH)	average	102	0.26
	worst	102	0.26
TOPPERS on SPUMONE (SH)	average	367	0.92
	worst	1582	3.96
native TOPPERS (x86)	average	124	0.05
	worst	171	0.07
TOPPERS on L4 (x86)	average	5867	2.45
	worst	27270	11.39

Table 4.3 shows the time required to build Linux kernel on native Linux and modified Linux executed on top of SPUMONE together with TOPPERS. TOPPERS only receives the timer interrupts each $1ms$, and executes no other tasks. The result shows that SPUMONE and TOPPERS impose overhead of 1.4% to Linux performance. Note that the overhead includes the cycles consumed by TOPPERS. The result shows that the overhead of the virtualization to the system throughput is sufficiently small.

The average delay of TOPPERS on SPUMONE is 0.25 μs (102 cycles) against the average delay of TOPPERS on L4 which is 2.45 μs (5867 cycles) shown in Chapter 3. The result of L4 is measured on the x86 architecture, thus the comparison might not be fair, however it clearly shows that SPUMONE's interrupt delay is smaller in the wall clock time. The result shows the benefit of hosting both the hypervisor and the guests in the same privileged level: omitting some save/load of the processor registers.

Table 4.3: Linux kernel build time

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

4.4.2 Engineering Cost

We evaluated the engineering cost of reusing the RTOS and the GPOS by comparing the number of modified lines of code (LoC) in each OS kernel. Table 4.4 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 are them. We did not include the LoC of utility device drivers provided for communication between Linux and RTOS or Linux and servers processes because it depends on how many protocols and how complex those are implemented.

Since we could not find RTLinux, RTAI, OK Linux for the SH-4A processor architecture, we evaluated them developed for the Intel architecture. OK Linux is a Linux kernel virtualized to run on the L4 microkernel. For OK Linux, we only counted the code added to the architecture dependent directory `arch/l4` and `include/asm-l4`. The comparison would not be fair in a precise sense, however as the table shows, it is clear that our approach requires significantly small modifications to the Linux kernel. This result is achieved because we are executing OS in kernel mode.

Table 4.4: The total number of modified LoC in *.c, *.S, *.h, Makefiles

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

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

We measured the effect of loads on Linux to the dispatch delay of a TOPPERS periodic task. We compared two methodologies to reduce the interrupt response time interference by Linux. One is the interrupt priority

A periodic task runs every $1ms$. It is sampled 100,000 times during the measurement. The dispatch delay is the time spent from the interrupt trigger till the periodic task starts its execution. Only the periodic task is executed on TOPPERS which means no other task on TOPPERS will prevent the execution of it. Figure 4.6 proves that no task on TOPPERS affects the cycle of the periodic task.

Figure 4.7, 4.8 compares the distribution of the timer interrupt delay without and with IPL separation (described in Sec. 4.3, Fig. 4.4) under continuous `stress` on the processor utilization of Linux. The worst case without the IPL separation marked 17 μs, with the IPL separation 13 μs; the worst case delay slightly reduced.

Figure 4.9, 4.10 compares the distribution of the timer interrupt delay without and with IPL separation under continuous `write()` to a CF card filesystem. We executed `stress` as a workload on top of Linux. The measurement with filesystem load shows a maximum delay of $111\mu s$ without the IPL assignment. With the IPL assignment, this

delay is decreased to $34\mu s$. Comparing this result with the measurement done by [13], with 1.8GHz Athlon processor which shows maximum delay of a few hundred μs, we can see that our measurement with 400MHz SH processor achieves fairly small dispatch delay.

Figure 4.11, 4.12 compares the distribution of the timer interrupt delay without and with virtual core migration under continuous `write()` to NFS share file system. The measurement without virtual core migration shows the maximum overhead of 96 μs. With migration is enabled, the maximum delay is 39 μs.

Figure 4.13, 4.14 compares the distribution of the timer interrupt delay without and with virtual core migration under frequent IPC load on top of Linux. The IPC load is generate by `hackbench`, which is modified to acquire clock cycles from a device file which returns correct count independent of the processor utilization of the RTOS. The delay measured without virtual core migration numbered 3770 μs. This is because the interrupt assigned to the RTOS is blocked by the spinlock mechanism of Linux. With migration enabled, the interrupt delay is 44 μs.

The overall measurement shows the IPL assignment and the virtual core migration mitigates the effect of lock disabling performed inside the Linux kernel. Even though this measurement only shows the statistical maximum interrupt delays, it is clear that they can reduce the average interrupt delay.

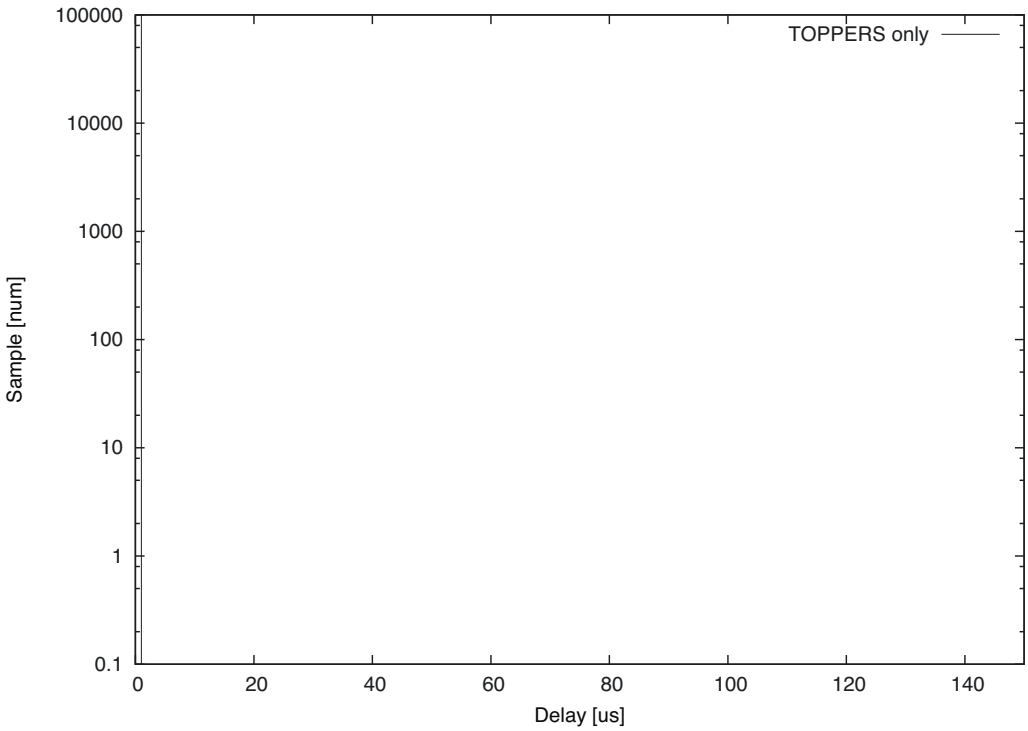


Figure 4.6: The dispatch delay of $1ms$ periodic task on native TOPPERS)

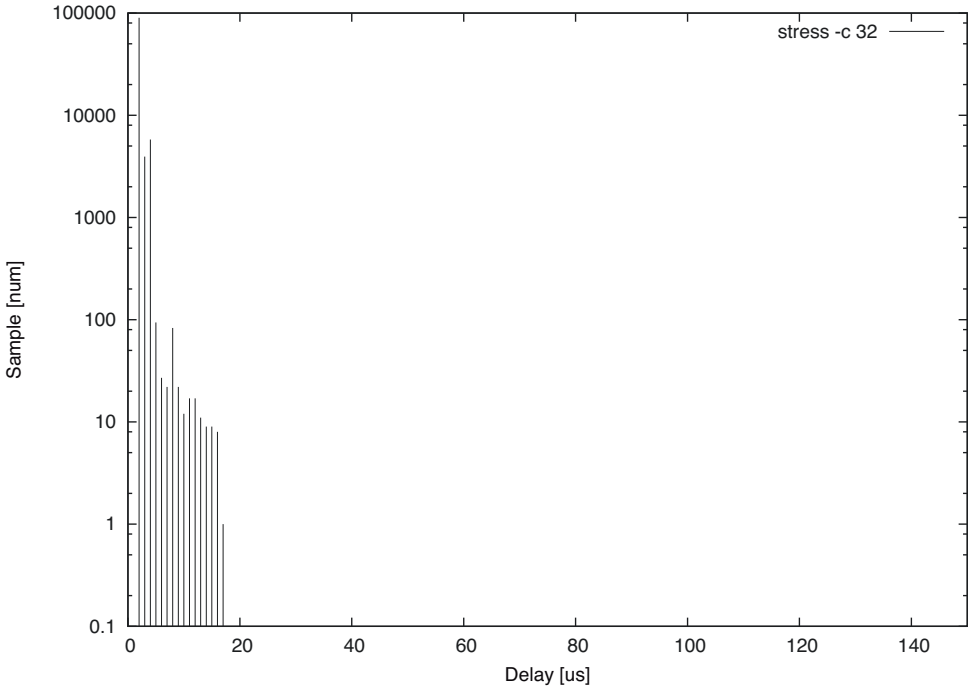


Figure 4.7: Dispatch delay (CPU `stress` on Linux without IPL modification)

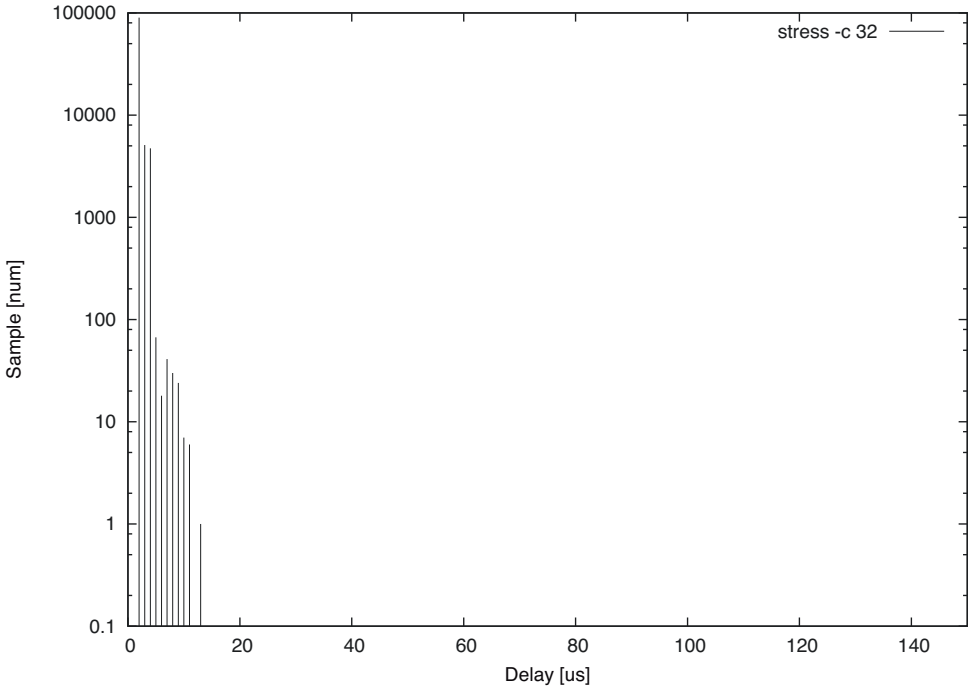


Figure 4.8: Dispatch delay (CPU `stress` on Linux with IPL modification)

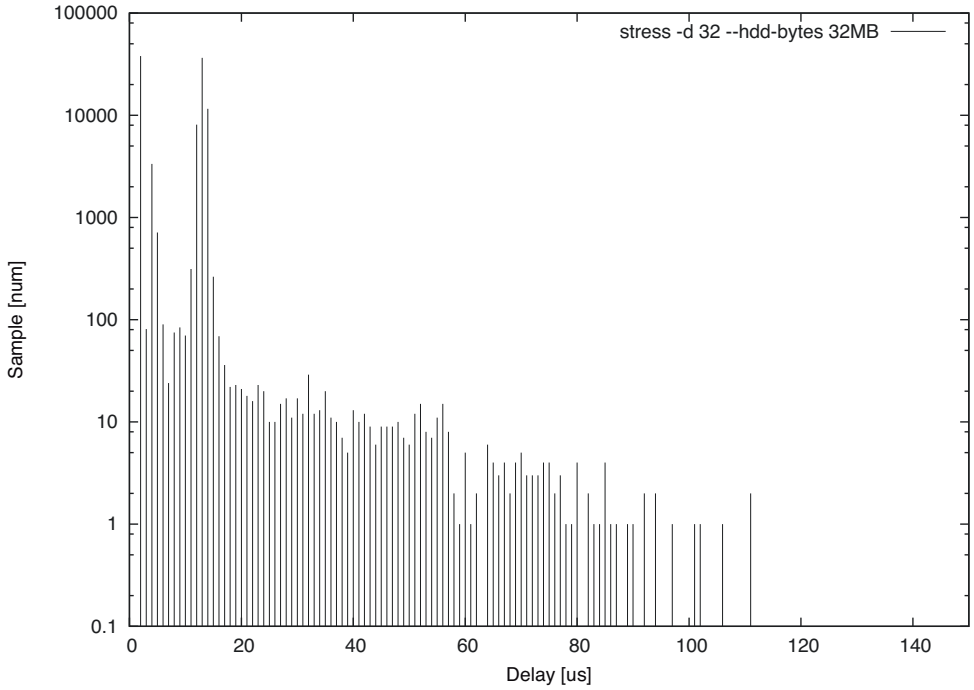


Figure 4.9: Dispatch delay (CF read/write `stress` on Linux without IPL modification)

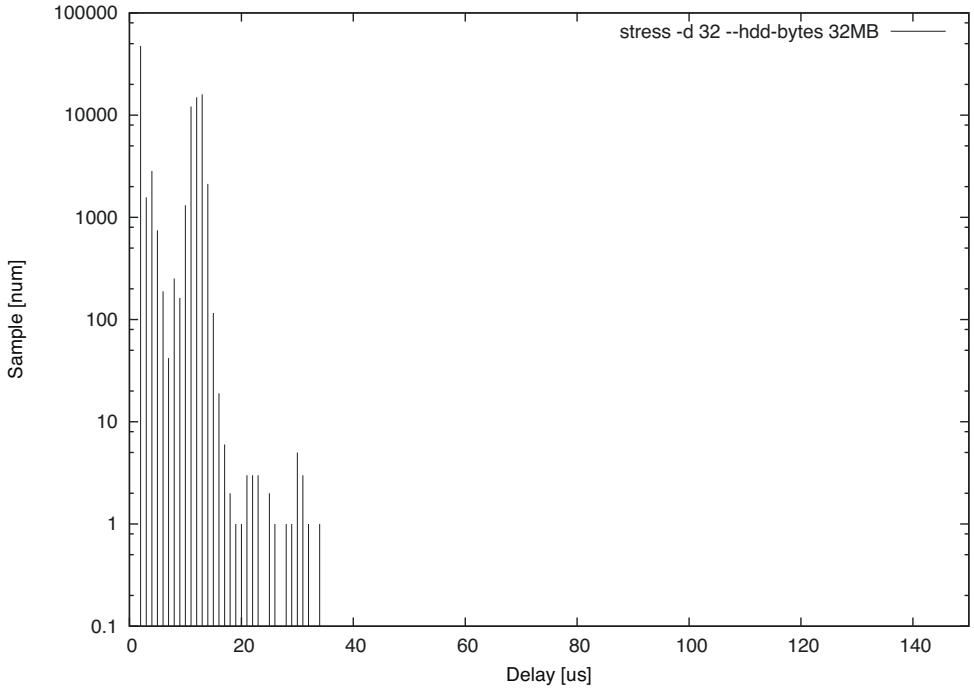


Figure 4.10: Dispatch delay (CF read/write `stress` on Linux with IPL modification)

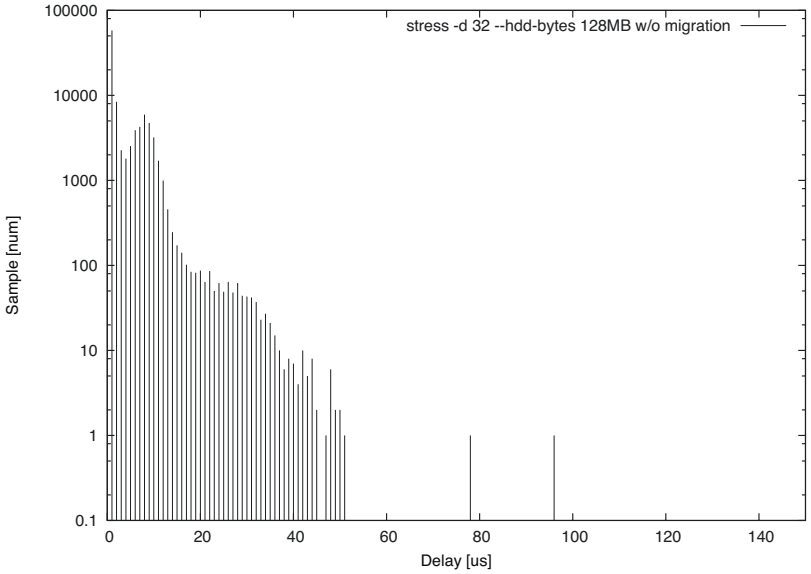


Figure 4.11: Dispatch delay (NFS r/w `stress` on Linux without virt. core migration)

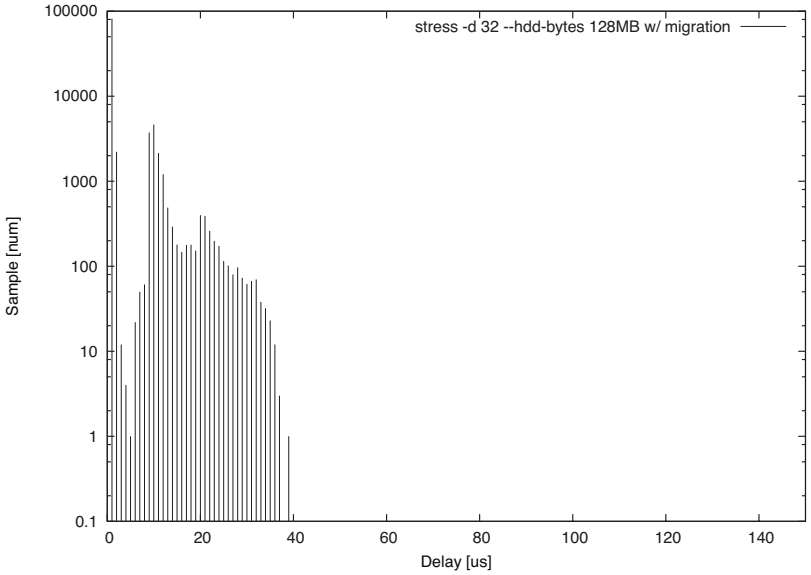


Figure 4.12: Dispatch delay (NFS r/w `stress` on Linux with virt. core migration)

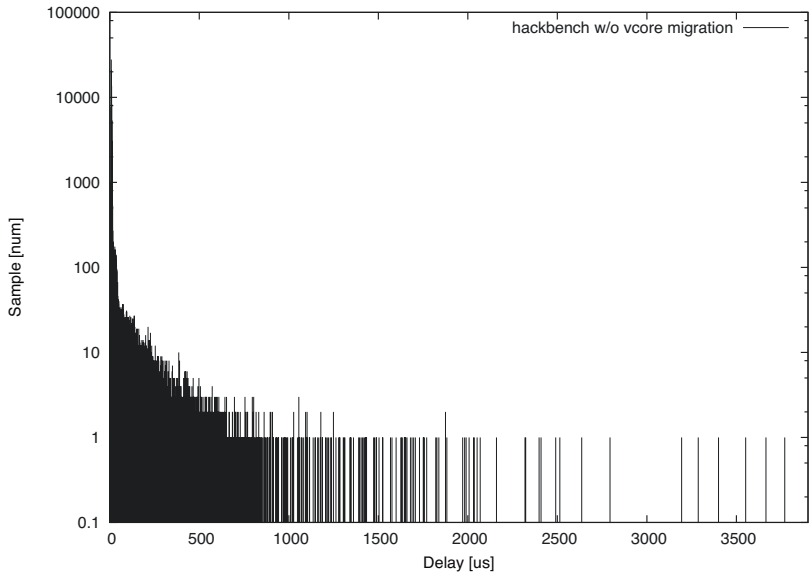


Figure 4.13: Dispatch delay on SMP (frequent IPC on Linux without virt. core migration)

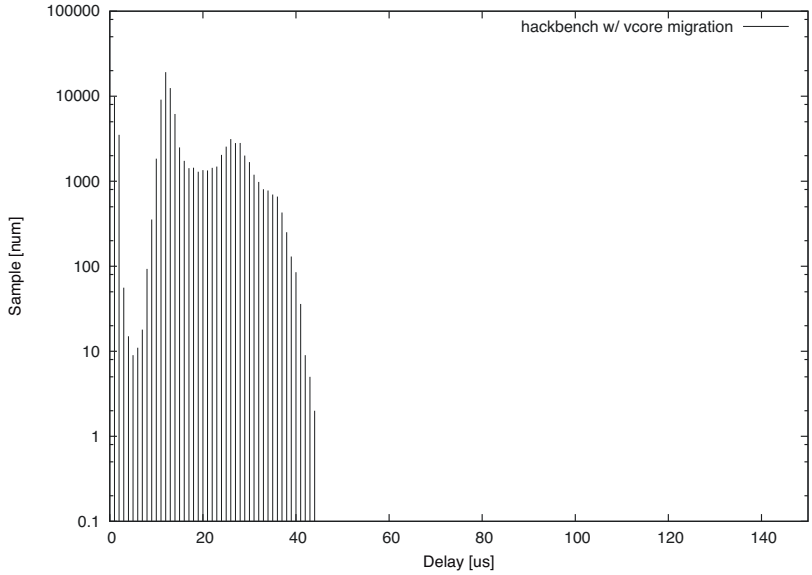


Figure 4.14: Dispatch delay on SMP (frequent IPC on Linux with virt. core migration)

4.4.4 The Effect of TOPPERS Periodic Task Load to Linux Throughput

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

Figure 4.15 shows the total score of Dhrystone benchmark executed one on each core. The bar at the left end shows the score of evaluation done with Linux executing by itself on top of SPUMONE with limited usage of three cores. core 1 and core 2. As long as the utilization of the periodic task grows, the score of Dhrystone degrades. At load of 90%, the result gets close or less than to three dedicated core configuration. The result shows the overhead of virtual core migration is kept in reasonable value.

In contrast, Fig.4.16 shows the score of hackbench, a benchmark which evaluates the scalability of the number of cores. The execution time of hackbench is increased with the virtual core migration enabled. This is entailed by the frequent system calls invoked during the benchmark, which triggers virtual core to migrate among physical cores many times.

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

The score of hackbench quickly drops along with the increase of load on TOPPERS. The source of this performance reduction is the phenomenon known as lock holder preemption. Related work has been tackled this phenomenon by giving longer time slice to a virtual core which is holding a spin lock [55, 64]. Those methodologies cannot be adapted to our platform because the remaining lock-holder on a processing may cause RTOS deadline miss. In addition the application invokes frequent system calls or interrupts, the overhead of virtual processor migration increases.

4.5 Related Work

Various approaches are proposed to balance real-time responsiveness and rich functionalities on a single platform. One of the approaches is modifying a GPOS to support real-time responsiveness. The real-time patch is a modification to a plain Linux kernel to support kernel preemption[45]. It achieves a few hundred μs latency[13], but still the result is

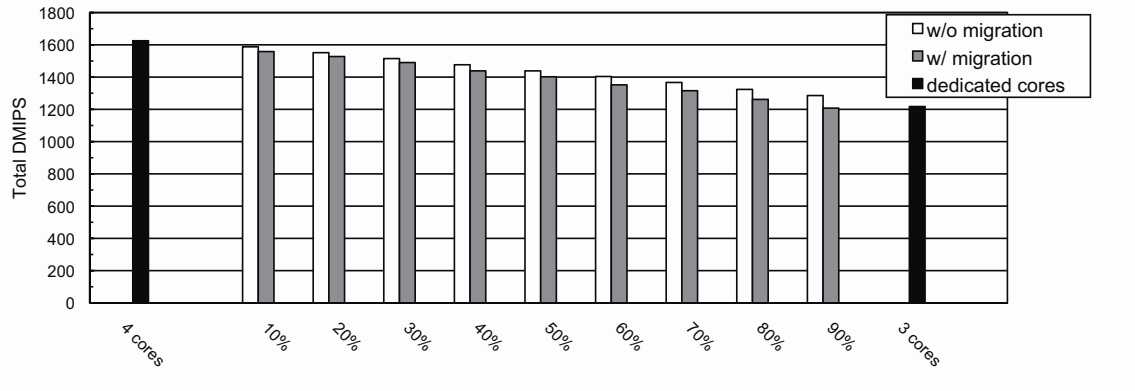


Figure 4.15: The effect of load on TOPPERS to Linux's DMIPS score (y-axis in DMIPS, larger is better)

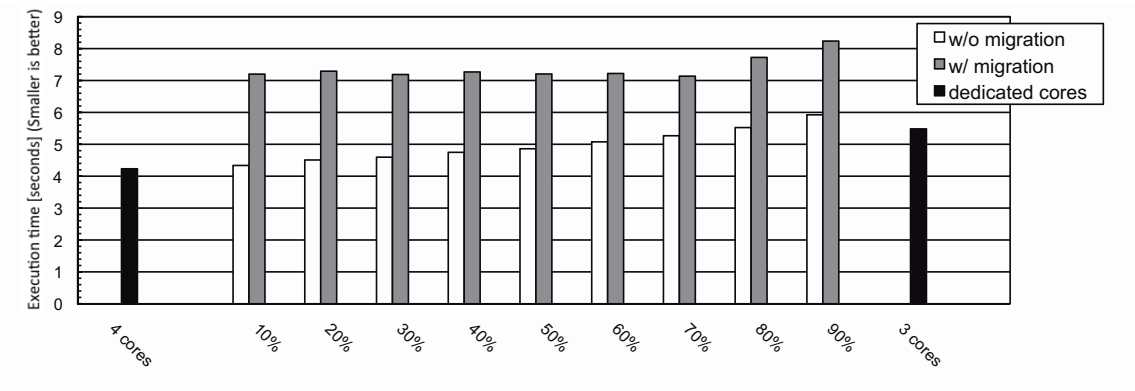


Figure 4.16: The effect of load on TOPPERS to Linux's hackbench (y-axis in seconds, smaller is better)

slower by a factor of ten comparing to typical RTOSes. Even though the mechanism is potentially capable of achieving real-time responsiveness, it could be easily spoiled by a bad-mannered device driver, which holds a lock for a long period. Porting software from an RTOS to Linux would increase the risk of implementing such drivers, because of the difference of programming models between the RTOS and Linux or the developers being unfamiliar with programming on Linux. In addition, porting all the software from the RTOS to Linux would impose substantial engineering cost.

Another approach, known as the hybrid system, is to link an RTOS with a GPOS. RTLinux and RTAI replace the Linux hardware abstraction layer with their own version of RTOSes [66, 44]. Those RTOSes would be executed in kernel mode together with the Linux kernel. This design is similar to the design of SPUMONE which puts the virtualization layer and kernels into the same privileged level. The interrupt response time would only be a few μs, which is comparable to typical RTOSes. However those microkernels only support their original programming interfaces, which prevents the straight-forward reuse of

some real-time software developed for traditional RTOSes. In contrast SPUMONE can run RTOS kernel with a few hundred LoC of modification to the kernel. Linux on ITRON is an alternative method to RTLinux and RTAI, which replaces the Linux hardware abstraction layer with an existing RTOS, μITRON[59]. This architecture enables the system to reuse both the software developed for Linux and μITRON. The hybrid system provides high real-time responsiveness comparable with an RTOS with reasonable engineering cost by reusing existing GPOSes. However considering another combination of an RTOS and a GPOS would impose redesigning the hybrid system again from scratch. Because it is usual for manufacturers to leverage diverse OSes, this engineering cost would be problematic. SPUMONE removes the dependency between the hosted RTOS and GPOS in order to reduce this cost of adpoting various OSes.

A hypervisor is another technology focusing on accommodating an RTOS and a GPOS into a single embedded device without modifications or with just minimal modifications to the OS kernels [31]. A hypervisor provides a virtual hardware interface which is identical (or almost identical) to some real hardware and isolation between virtualized guest OSes. A hypervisor supporting full-virtualization exposes a virtual hardware interface identical to a real hardware interface. OSes can be executed without any modification on full virtualization. However, implementing full virtualization complexifies the design of the hypervisor itself or requires hardware support for virtualization. Unfortunately hardware support for virtualization is still an unfamiliar feature for embedded system processors. This motivates embedded system hypervisors to use para-virtualization for their system design, like L4 did. However, the engineering cost required for para-virtualizing a guest OS kernel is also problematic for manufacturers. In addition, switching privilege level between a guest OS and a hypervisor will entail performance degradation.

In order to achieve low engineering cost while not penalizing performance, our virtualization layer executes OS kernels and itself in privileged mode. This also contributes to reduce the engineering cost of modifying OS kernel, because majority of privileged instructions can be executed by a processor directly and only minimal set of instructions needs emulation. Furthermore, The virtualization layer multiplexes only minimal hardware resources, while other resources are exclusively assigned to each OS by simply modifying each OS kernel not to access the same devices.

There are some researches on how to design a scalability OS on multicore and many-core processors. Corey [20] is an OS which allows applications to explicitly specify the assignment of data structures among cores. This hints the kernel to schedule processes to improve cache locality. Multikernel is a experimental OS kernel which exploits multi-core and many-core processors parallelism by constructing the system with distributed model [18]. Our basic design is similar to the one of theirs. However our contribution is to let a composition kernel to reuse software stacks developed on top of existing OSes while retrieving real-time responsiveness.

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

4.6 Summary

This chapter we introduced our light-weight virtualization methodology which achieves low overhead and low engineering cost for constructing multi-OS embedded systems. In addition we evaluated the real-time responsiveness of a RTOS running concurrently with Linux under various loads. We proposed two methodologies to mitigate the performance interference from Linux to the RTOS; the IPL assignment and the virtual core migration. The evaluation shows that our methodologies reduced the interrupt delays. Especially on multicore system, Linux sharing a core with a RTOS increases processor intensive applications. However with application triggering frequent system calls may lose its performance due to the frequent virtual core migration among physical cores.

Chapter 5

Core-local Memory Assisted Protection

In the previous chapter we introduced a light-weight multi-OS architecture without support of a hypervisor. In order to reduce the overhead of using a pure-hypervisor, our method runs a RTOS and GPOS kernel within the same privileged level. However, although the security between the RTOS and GPOS is not necessary, it is useful to provide an isolated execution environment that enables running security monitors unexploitable from guest kernels. Instead of relying on approaches based on traditional pure-hypervisors, we propose a novel approach that leverages some architectural features. We discuss a minimal set of hardware functionality to run security monitors safely. In addition to the hardware functionality, we proposed a software method to keep the integrity of the monitors by tracking the identity of their memory pages.

5.1 Background

It is crucial for modern operating systems to protect not only applications but also itself from malicious attacks. Especially, rootkits target a kernel to hide its malicious activities. This makes security monitors running in a user-level difficult to detect attacks, since monitors relies on the integrity of the underlying kernel [42]. For instance a rootkit may compromise the kernel's data structures so that it would not appear in a list of processes, or prevent the monitor to be scheduled. The problem inspired some researches to migrate monitors *outside* the kernel with help of virtualization technologies.

With help of a virtualization technology, monitors can run within a hypervisor or within a virtual machine instance. The monitor runs in a higher privileged than the monitored virtual machine (we call it a target system) or an independent virtual machine that is isolated by means of address space translation. However there are some shortages on the monitoring with hypervisors.

- As hypervisors become more functional, their code sizes are increasing. The large code size of a hypervisor makes it prone to vulnerabilities. Now the monitor's safety is depending on the underlying hypervisor, similar to the dependency between user-level applications and the underlying kernel.

- Furthermore the large code size makes hypervisor's behavior unpredictable, consequently makes it difficult to schedule virtual machines in real-time. As we discussed in Chapter 3 and Chapter 4, we claim hypervisors that use additional privileged level does not fit use in embedded systems.

In this chapter, we introduce a method to protect monitoring system without support of hypervisor. We propose a processor equipped with a memory area that is accessible only from a specific processor core. This memory area is designed to be private to the core, which cannot be reconfigured accessible from the other cores. This is called a core-local memory. We give a few additional limitation to the functionality of the processor to make the local memory inaccessible even from privileged programs like OS kernels and maliciously promoted rootkits, thus the monitor runs safely within its private space.

In addition we propose a method to virtually extend the space available on the core-local memory; we call the method the secure paging. The idea of the secure paging is similar to the disk swap mechanism of GPOSes. Instead of swapping data between memory and disk, our method swaps memory pages between local memory (private to the core) and the main memory (shared among cores). Those pages stored in the shared main memory can be corrupted by the other cores. In order to track their integrity, the secure paging calculates the hash values of those pages.

The combination of the core-local memory and the secure paging allows running monitors safely without help of hypervisors. In addition it keeps the integrity of the monitoring system even from the attack privileged code, especially from maliciously promoted malware.

5.2 The LLM Architecture

The secure pager's method is twofold. In this section, we introduce the hardware feature necessary for executing security monitors, which we call the LLM architecture. In order to utilize the core-local memory for securely running a monitor, we need to ensure the code of the monitor to be executed in the local memory and to prevent it from being altered or halted by the malicious programs running on the other cores.

A symmetric multicore processors (SMPs) are equipped with several cores that have symmetric access to the shared main memory. In order to reduce their access latency to the main memory, each core is equipped with a small memory region to store a portion of main memory, usually known as a cache. The caches on different cores may refer to the

same region of the main memory. This might cause the data inconsistency among cores. SMP supports data cache coherency mechanism to ease the management of caches. If a core modifies shared data, the copies of it residing in the other cores are updated or their values are disabled.

The increasing number of cores motivates redesigning processors to support loose coherency among cores. Along with the expansion in the number of cores, the area on a chip for inter-core communication (network) may drastically increase. A strict cache coherency model is expensive to be supported in the upcoming many-core architectures. There are some real-world processors that support local memory. Hitachi's RP1 [67] and RP2 [37] are equipped with core-local memory . Intel has announced SCC, which has sets of dual cores sharing a non-coherent cache area [10, 11, 12]. Each core is equipped with a small portion of memory that is private to it. The contents of the core-local memory can be accessed only by the core that has the memory. The core-local memory is isolated from the other cores. In addition to the private access, some real-world local memories supports remote access from the other cores. Within a few cores, the remote access may be reasonable but with dozens of cores, connecting the cores will dominate large area on the die. Therefore the LLM architecture advocates the design of isolating core-local memory. The size of the core is typically a few hundred kilobytes. Note that core-local memory is not synchronized with the main memory like a cache memory. The program should explicitly transfer data between the core-local and main memory.

In the normal SMP processor, cores have equal privileges. Any core can force other cores to branch to a specified address. This feature is used to reschedule a program from one core to another, also to start the cores during the system's bootstrap and to recover a core that is stopped in infinite loop. The LLM architecture breaks this assumption; one core runs in a higher privilege than the others. In this dissertation we call the privileged core *core0*, and the others *coreU* in the track of virtualization terminology. In the LLM architecture the core that hosts a security module (core0) gains a higher privilege than the other cores (coreU). The privileged core has the following features:

- First the core is ensured to be booted in advance of the other cores. With this feature, the privilege core can safely initialize itself without being compromised by the other malicious applications. The computer starts the initial code with core0. The BIOS with *Trusted Platform Module* (TPM) features can enforce core0 to execute verified program. Core0 contains the security monitor into the local memory to the local memory and then start the other coreUs. The usage of this features is described in detail in Section 5.3.

- Next, it cannot be reset by the other unprivileged cores. This ensures the monitor not to be deactivated by malicious code, and not to execute malicious code that tries to access the core-local memory.

In addition to the hardware extension, some critical data structures should be managed carefully by software in order to pin down the security monitor on the privileged core.

- The security monitor should place interrupt and exception vector tables in the local memory so to make it immutable from coreUs. Interrupts and exceptions can potentially change the execution path of a core. For instance, if a malicious program (running on coreU) modifies the interrupt vector table contained in the main memory, the attacker can modify the vector's entry to point some malicious code. By protecting the table, core0 can safely handle the interrupts sent from coreUs.
- Page tables must be contained in the local memory. If page tables are contained in the shared main memory, exploited to coreUs, malicious applications may modify them to let the monitor execute arbitrary code.

Unfortunately we don't have a real implementation of the LLM architecture now. However the strength of our idea is that it is applicable to the existing SMP architecture with some slight modifications and without updating the core instruction set architecture. The core-local memory is already implemented on some hardware. The remote access can be disabled easily by disconnecting the data path among cores; in other words it is more straightforward to design an SMP with core-local memory that does not support remote accesses. For the privileged core we need to modify the interrupt controller of the processor to simply ignore the signals that reset and halt core0. Again here we claim that the LLM architecture is a reasonable extension to the existing SMP architecture.

5.3 The Secure Paging

In the previous section we introduced the LLM architecture which executes a program stored in a core-local memory safely without support of a hypervisor. However, by the LLM architecture itself, the size of the local memory bounds the size of the monitor that can be executed on top of it. It is inflexible to limit the size of the monitor by the amount of an available hardware resource. For instance, the size of local memory would vary from processor to processor. In addition, the size of the monitor might be significantly larger than the size of the local memory; the monitor may refer to some large data structures that describe the behavior of malicious applications (e.g. virus definitions). In order to mitigate the size limitation of the LLM architecture, we propose the *secure paging* that virtually extends the size of a core-local memory. s

The secure paging dynamically swaps memory pages between a core-local memory and a shared main memory. The idea is similar to paging: swapping memory pages between memory and a disk storage. The monitor's pages are mostly stored in the main memory, and loaded into the core-local memory when they are accessed. This way, we

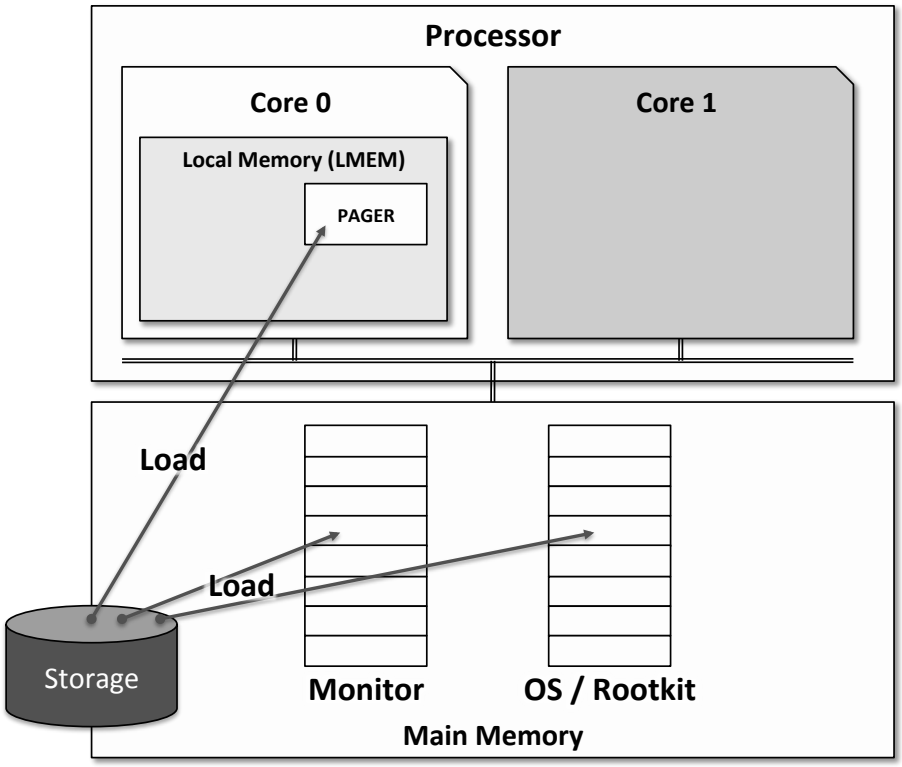


Figure 5.1: Loading the pager, the monitor and the target OS.

can assign memory space larger than the core-local memory. However the pages located in the shared main memory, that are accessible from all cores including coreUs, should be protected from malicious attacks. The secure paging prevents executing modified program by tracking the integrity of each memory page.

We explain the mechanism of the secure paging step by step. A boot loader first loads an initial program from a storage to the local memory. We assume the integrity of the initial program is guaranteed with the help of the LLM architecture described in Section 5.2. A small strip of code is copied into the core-local memory and executed there for the remaining lifetime of the system. We call it the *secure* pager or simply the *pager*. The pager then loads the kernel image of the monitor and target into the shared main memory. For explanation, we take the setup shown in Figure 5.1, core0 executes the monitor and core1 executes the target OS. Core0 is a privileged core and core1 is unprivileged core, coreU. At this point core1 is not started so there is no malicious activities that interfere the setup of the pager. The figure is a simplified model and the does not represent the real memory allocation and consumption.

The pager, first calculates the hash value of each page that belongs to the monitor (Figure 5.2). Those hash values are stored into the hash table allocated in the local memory. A hash value is the digest of the corresponding page. It is infeasible to update the contents of the page without changing its hash value. Furthermore, It is infeasible

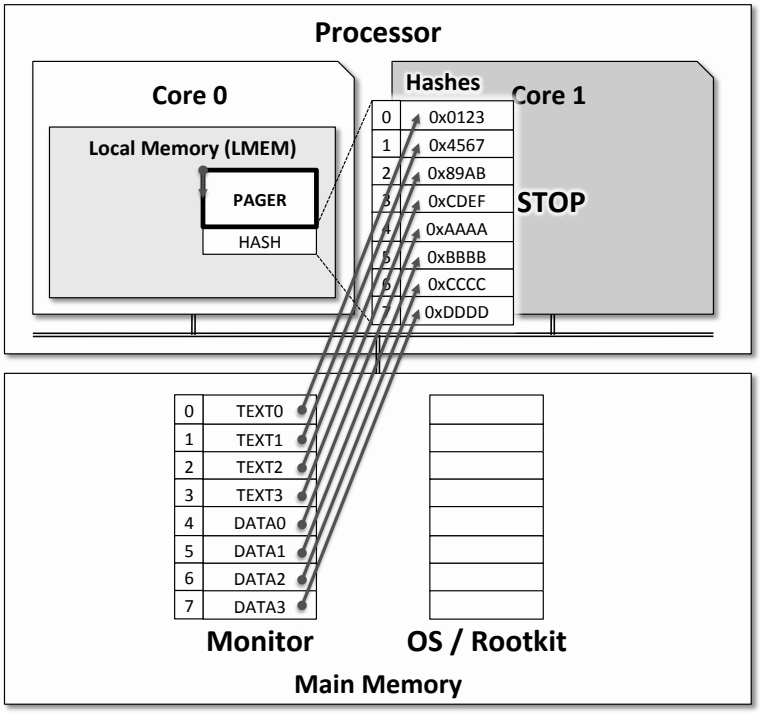


Figure 5.2: The pager calculates the hash value of each page. The target OS is not activated at this point.

to inject a meaningful code into a page without changing the hash value. Hence, if the hash value remains constant, the pager can consider the page is unmodified and contains a coherent data. It is up to the developer to select which hash algorithm. The developer should consider some trade-offs among cryptographic hash algorithms. The difficulty of the hash algorithm may improve the security of the system, however on the other hand it may add overhead to each swap-in to the core-local memory and swap-out to the share main memory. In addition larger size of a hash value consumes larger memory space in the core-local memory. The maximum memory footprint of the monitor is bounded by the size of the core-local memory that is allocatable to the hash table. We can mitigate this limitation by storing hash tables in the shared main memory and tracking the integrity of the table similarly to the pages of the monitor. Since this method complicates the algorithm of the secure paging, we focused on the experiment with the first basic idea and left the extension as our future work.

Next the pager initializes core0 to keep track of the activity of the monitor (Figure 5.3). The pager uses page faults to interpose the memory accesses of the monitor. The pager must allocate page tables and trap vector tables in the core-local memory in order

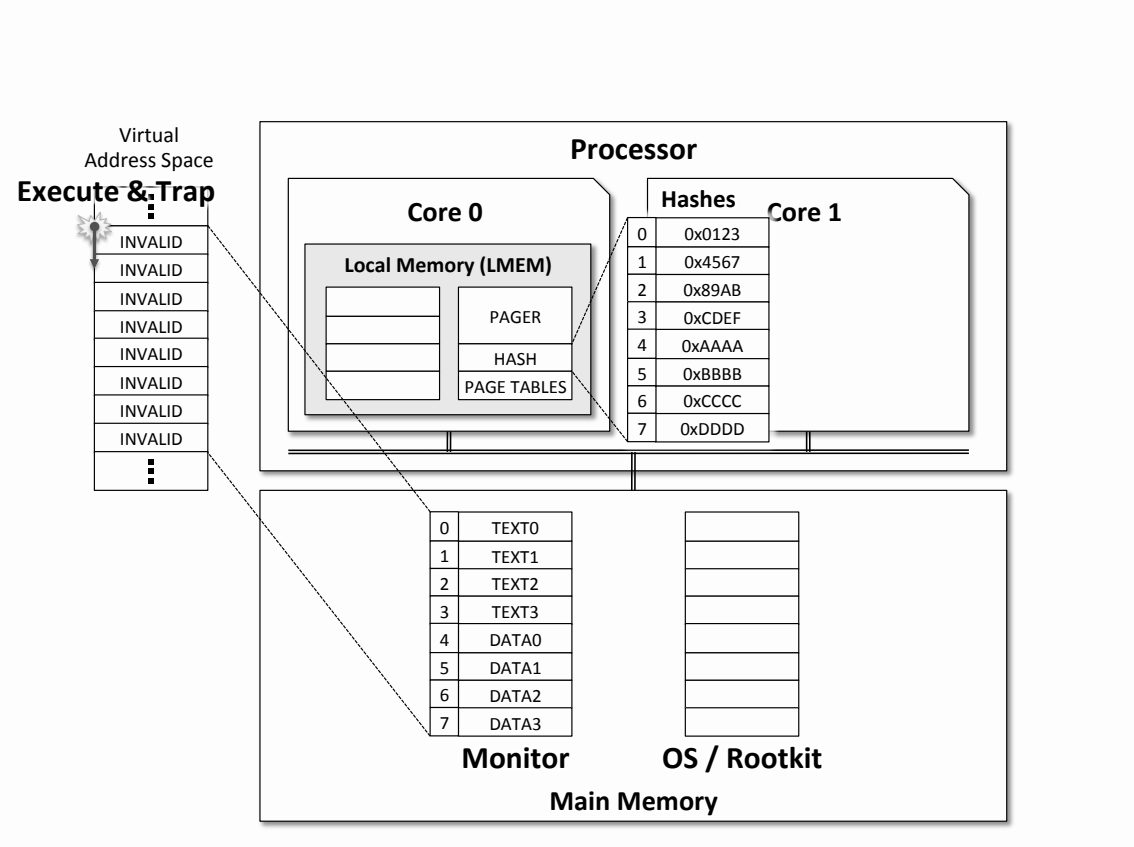


Figure 5.3: Execute the entry of the monitor and trap into the pager.

to protect these critical data structures tampered by malicious programs. Otherwise, if a page table remains in the shared main memory, a malicious application can modify the entry in the table to map a tampered text into the monitor and overwrite handler with its own function. With the Intel architecture, which we used in the experiment, the pager also need to locate descriptor tables in the core-local memory.

In the example we assume that the monitor is executed in a single virtual address space which maps the physical pages of the monitor flatly to a corresponding virtual address. This is the most simple case for managing the monitor's address space. If the monitor is a simple program that runs in a single address space, the pager can occupy the MMU. If the monitor uses multiple address spaces or run under an OS that provides dedicated address space for each process, the pager must interpose operations to MMU and page tables in order to enforce the limitations, not to allocate page tables in the shared main memory and not to map pages in the shared main memory directly to the monitor's address space.

Since all the entries of the page table is initialized as unmapped, the monitor immediately triggers a page fault at its entry point. Receiving the page fault, first the pager copies a corresponding page, which is stored in the shared memory, into the buffer

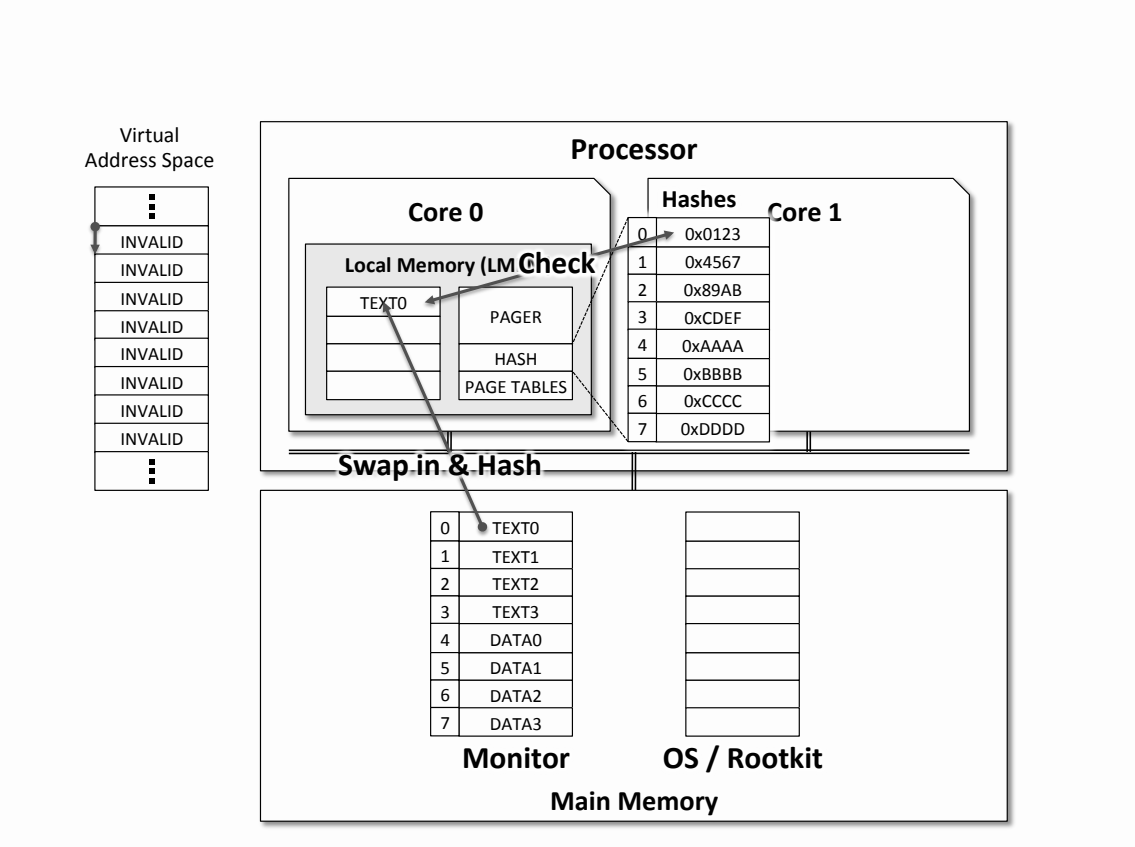


Figure 5.4: The pager swaps in a page while calculating its hash value, and checks if the page is altered.

allocated in the local memory. Next the pager calculates the cryptographic hash value of the page loaded in the buffer. In Figure 5.4, the pager loads the first text page of the monitor into the core-local memory. The hash value of the text page is calculated and compared with the value stored in the hash table. In order to calculate the hash value safely, the pager should first copy the page into the local memory and then calculate its hash value. If the pager directly calculates the hash value of the page in the shared main memory, the page could be tampered before it is copied to the buffer in the core-local memory. Depending on the cryptographic hash algorithm, the pager can calculate the hash value along with copying the page. In this case, tampering the page during the copy may change the resulting hash value, which can be detected by the pager as an malicious update.

The pager tags the loaded page with the physical address of the page. This physical address is used to index the corresponding entry of the hash table when writing back the page to the shared main memory.

As shown in Figure 5.5, if the calculated hash value matches the pre-calculated value, the pager maps the copied page to the address space and resumes the execution.

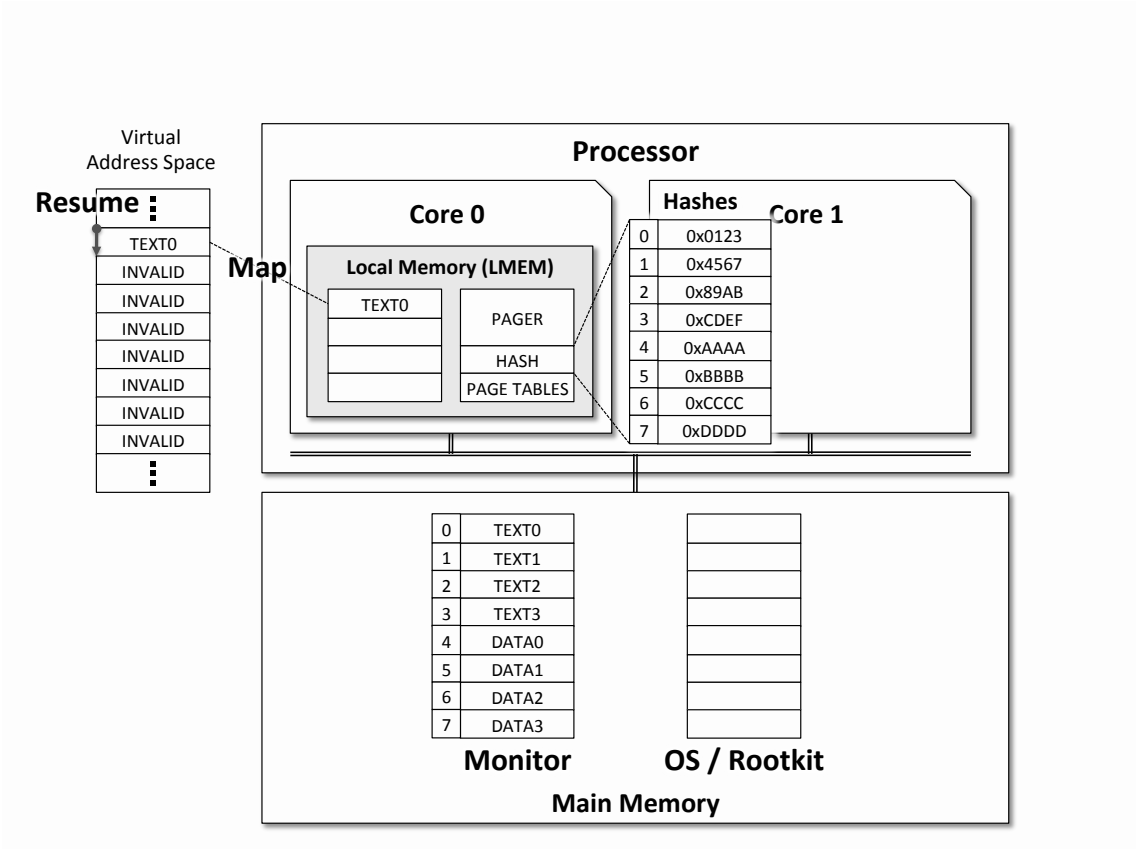


Figure 5.5: The pagers maps the checked page into the address space.

When the monitor accesses another page that also triggers a page fault, it is similarly copied, hashed, checked, mapped and resumed. The pager and the monitor repeat these phases and keeps the integrity of the monitor during its execution.

Note that the pager only keeps track of the integrity of the monitor; not the target OS. We would like to remind that the pager is in charge of checking the integrity of the monitor and the monitor is in charge of checking the integrity of the target OS. This dependency is similar to the dependency of the hypervisor based monitoring, which the integrity of the monitor is depending on the underlying hypervisor, and the monitor is checking if the OS is exploited or not. It depends on the monitor's ability to detect the malicious attacks against the target OS. We are not proposing a novel OS monitoring but instead introducing a method to securely host a monitor and a target OS without support of a hypervisor. The difference is that in the secure paging, the target OS cannot gain a privilege to access the pager since the program in the core0's local memory is isolated by means of the LLM architecture. In the hypervisor based approach, the target OS kernel dynamically interacts with the hypervisor that is vulnerable.

If a page fault is triggered when there is no free space in the core-local memory, it needs to evict a page back to the shared main memory (Figure 5.6). The pager calculates

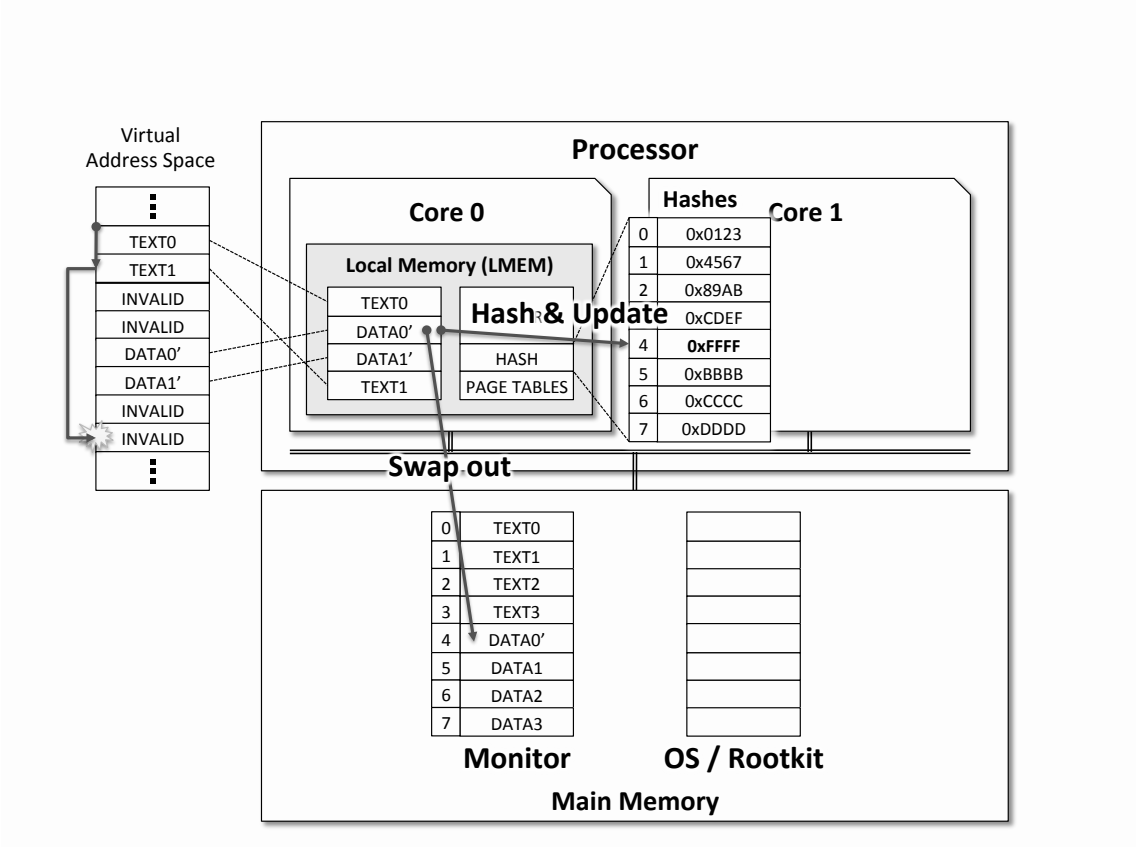


Figure 5.6: The pager tries to evict a page contained in the local memory to the main memory.

the hash value of the page and updates the value stored in the hash table. As we mentioned above, copying and calculating the page can be done at the same time. We can adopt several optimizations to improve the performance of the page eviction. First, the pager can omit updating the hash value and writing back the page to the shared main memory if the page is unmodified. The pager can retrieve page's update from the access bit of a page table entry. Usually the text area of the program is constant, thus it is freed immediately without hashing and copying on its eviction. Next, the algorithm of selecting a page to evict is also crucial to the performance of the secure paging. For the general case LRU algorithm might suit our purpose. If the pager can predict the behavior of the monitor, the pager can optimize its decision to select a free page. Furthermore developers can modify the monitor to align its functions to the page boundary and reallocate some frequently accessed data structures to fit into a single page. These optimizations are trade-off between performance and engineering cost. Optimizing the performance of a monitor running under the secure paging is a challenging topic, however we do not discuss it in detail in this paper, remain it as future work.

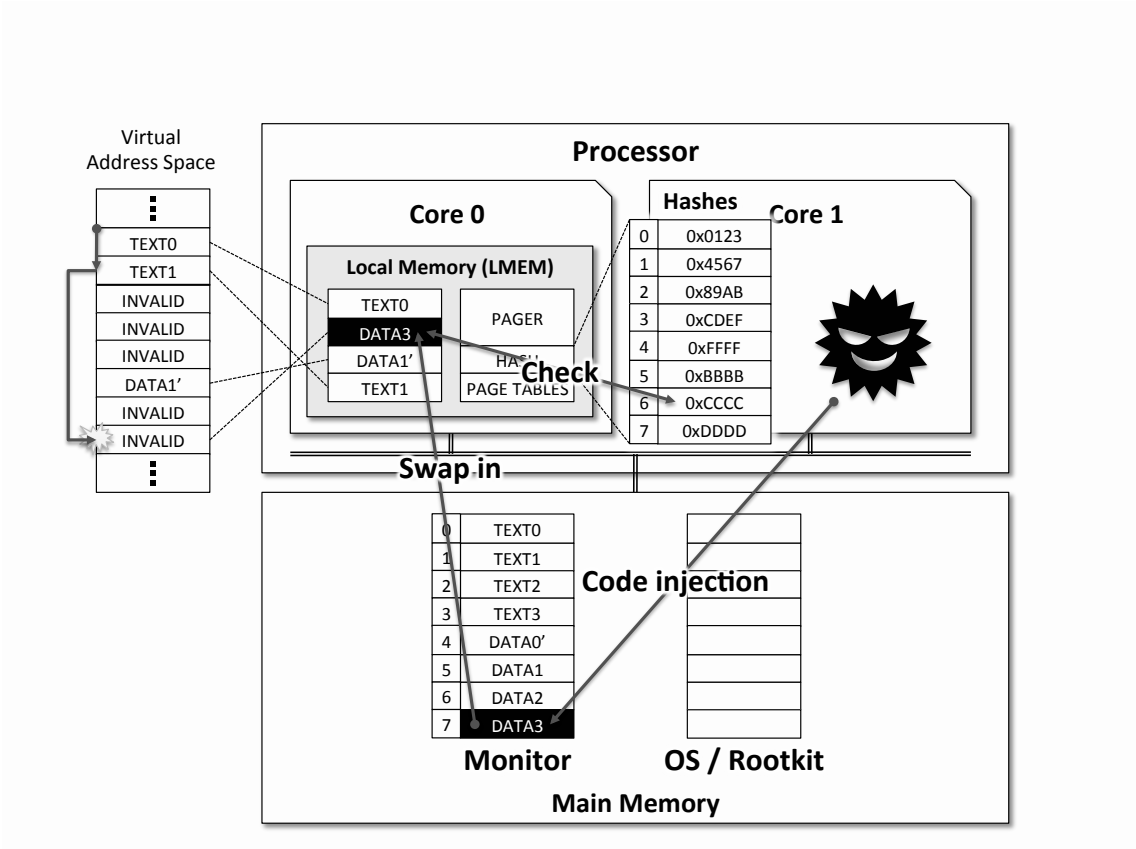


Figure 5.7: The pager detects a data page that is tampered by a malicious application running on the target OS.

5.3.1 Threat Detections

If the malicious code running on coreUs compromises the monitor's pages, the modification is detected when those page are next swapped into the core-local memory of core0. When the monitor detects an unmatched hash values, it will report security risk. Figure 5.7 shows an example that the attacker exploits the target OS and then trying to modify the data page of the monitor. The detection is delayed until the page is loaded in the core-local memory. This means the secure paging cannot detect attacks immediately. In order to reduce the windows of time to detect the malicious modifications, we can force the pager to periodically hash each page on the shared main memory. Surprisingly we found an eviction accidentally overwriting a tampered page with the original page. Also in this case, the page is recovered silently, thus the pager cannot detect the attack. Compared with some other intrusion detection systems, the secure pager's purpose is to guarantee the execution of untampered program. As we mentioned above, the pager is in charge of keeping the integrity of the monitor, but not the target OS. The monitor is in charge of detecting the rootkit in the target. We assume rootkits are more likely to attack some critical data structures of the monitor, that tend to be accessed frequently by the monitor,

which is detectable from the pager. Our method is free to select the method of monitor, like the hypervisor, therefore the accuracy of the monitor is out scope in this paper. We show our architecture is capable of hosting some existing monitoring methods in the next section.

5.4 Implementation

As a proof of concept, we applied our method to the two real-world OSes that runs on the QEMU machine emulator, without support of a hypervisor. As a target OS, we use Linux. As the monitoring OS, we use xv6. For some case studies we installed rootkits into Linux. xv6 runs a monitor that is in charge of checking the integrity of Linux. Without the support of the secure pager, the monitor can be directly modified by malware that has the kernel privilege. Since there are no real-world processor that fulfills the requirement of the LLM architecture, we assume a specific area of memory is assigned to the first core (core0) and not accessible from the others. For the ease of experiment, we developed our system on top of a machine emulator QEMU.

In order to run two kernels on a bare metal processor, we applied some modifications to both OSes. The method is similar the method of the logical partitioning [54].

The first modification is to reconfigure the OSes to allocate separate physical memory areas. In contrast to hypervisors, in the logical partitioning physical memory space is not virtualized therefore the OSes must voluntary select the range of physical memory to use (Fig.5.8).

Another modification is to limit the processors to use. In the logical partitioning, OS kernels would not migrate among cores. In our implementation, we modified xv6 to run on core 3 and Linux to dominate the remaining 3 cores. Core 3 is treated as core0. In the x86 architecture, core 0 always starts first. xv6 first runs on core0 than immediately migrates to core3. The initial steps are different from the theory however the xv6 can safely migrate to core3 without interfered by Linux.

In addition, some peripherals should be assigned exclusively. Linux is modified not to interfere xv6's serial communication. The serial communication is not mandatory because the monitor continuously accesses the target's memory and not the other peripherals. Malicious attack may reconfigure the serial channel and even more send a false messages. This could be prevented by encrypting the message so that the monitor and the administrator can detect altered messages. The safe way of assigning a communication channel to core0 needs some more limitations or additional hardware extensions.

The design of the secure pager (as described in Section 5.3) installs a modularized pager into the local memory. However in the experiment, we used xv6 kernel itself as the pager and its applications as monitor. In the modularized design the executive protects the entire monitor including the xv6 kernel. However in the modularized design, only

the exclusive is in charge of managing the address space. In the real implementation the xv6 kernel dynamically updates page tables. Therefore it makes hard to clearly split the functionality of executive and monitor. This might be resolved by leveraging the technique of shadow paging, which we did not implement in this dissertation. Some main data structures of the kernel is statically allocated in the local memory. The kernel assigns unmapped page to the applications. The corresponding page frames are allocated in the shared main memory. If the application triggers a page fault, the kernel copies the corresponding page frame into the local memory and calculates the hash value. If there is no free space available in the local memory, it swaps out a page to the shared main memory.

We modified the xv6 kernel to treat some critical data structures in special manners. First, we had to pin-down kernel stacks to the local memory in order to avoid nested traps. A kernel stack always resides in the local memory and would never be paged out. The Intel architecture implicitly saves some register values on the current stack when it receives interrupts and traps. If the stack is paged out, it crashes with double fault. This is the limitation of the Intel architecture we use in the experiment. In order to avoid this problem, in our preliminary implementation, we simply assigned a stack for each process. This allocation would waste some local memory area, however in our experiment only a few processes run on a kernel therefore we can assume the amount of memory to be small. Next, similarly, the page tables and descriptor tables must be allocated in the local memory. This is to avoid attackers to modify those structures to let core0 execute some malicious codes located in the shared main memory. For instance, modifying a page table entry can remap the xv6 kernel's page with a page that contains infinite loop to freeze the execution of the pager. These data structures that may reside on the shared memory and potentially may change the execution path of a processor core must be isolated by the local memory, which is not exploitable by means of software.

Linux that runs on coreUs almost equivalent to the normal Linux kernel. The only difference is the modification not to allocate the physical memory area that is assigned to the monitor. Except that, Linux is free to access any memory, storages and networks. We referenced the method in [54] to apply the modification to our Linux kernel.

5.5 Evaluations

In the experiment, we configured the size of the shared main memory as 1024MB. Linux is configured to use the first 800MB of the shared main memory. An kernel option is added to the kernel to control the memory usage of the Linux system. 512KB of memory area starting at 800MB is assumed as a local memory private to core 3. The remainder is assigned to the monitor, which is also accessible from Linux (Figure 5.8).

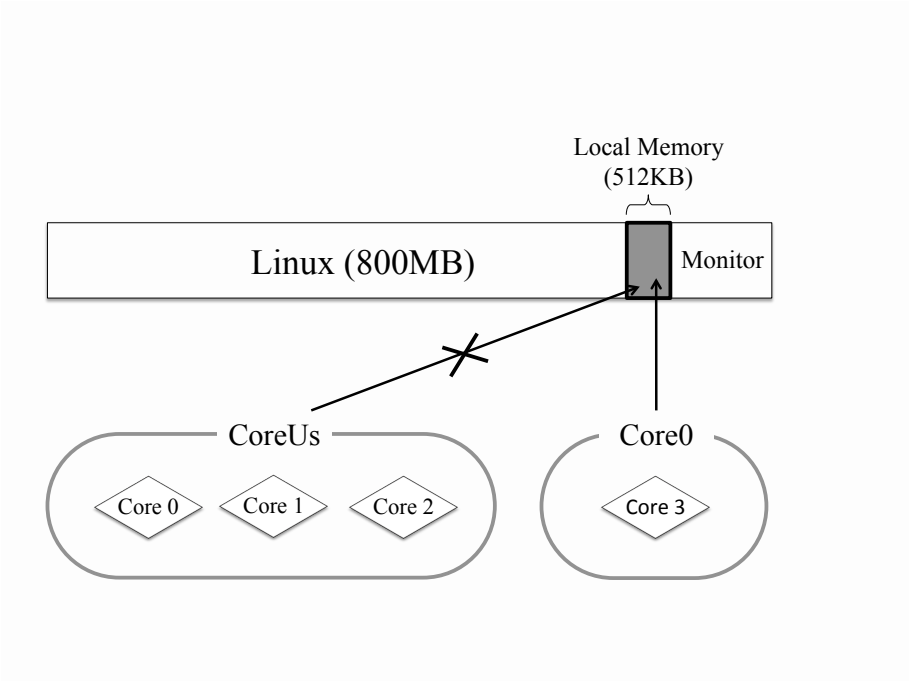


Figure 5.8: The memory layout of the pseudo LLM architecture.

5.5.1 Case study

As a case study we installed the following rootkits into the target system. These rootkits are obtained from PacketStrorm [5] if not specially mentioned. Some rootkits has been trimmed to support only the essential parts that compromise the target's data structures.

- **adore** is a rootkit that hides the existence of files and processes. It is provided with an interface to control the rootkit from the user-level applications. It achieves these functionalities by replacing system call table entries with its own version of system call functions. This malware is originally developed for Linux 2.4 but in our experiment we ported its key feature to Linux 2.6.26 which is our target.

  The monitor detects adore by monitoring the integrity of the system call table. The system call table of Linux should be unchanged through its lifetime. The monitor keeps the initial values of the system call table (saved to `sys_call_table`). During the target's execution, the monitor periodically acquires the latest copy of the system call table (copied to `sys_call_table_temp`) and compares it with `sys_call_table`. If there is unmatched entry, the monitor will report anomaly and halts the system. The following pseudo code shows the operation of the monitor.

```
void *sys_call_table[SYS_CALL_TABLE_LEN];
void *sys_call_table_temp[SYS_CALL_TABLE_LEN];
mread(sys_call_table, SYS_CALL_TABLE_ADDR, sizeof(sys_call_table));
```

```
while (1) {
  mread(sys_call_table_temp, SYS_CALL_TABLE_ADDR, sizeof(sys_call_table_temp));
  for (i = 0; i < SYS_CALL_TABLE_LEN; i++) {
    if (sys_call_table[i] != sys_call_table_temp[i])
      alert();
  }
}
```

- **knark** provides a file hiding functionality by replacing some system call table entries with malicious functions. We could detect its attack with the integrity checking described above.

- **adore-ng** provides some functionalities similar to adore but with a different mechanism; adore-ng compromises function pointers that are related to Linux's virtual file system. Especially in our experiment, we focused on adore-ng updating lookup function pointer, which saved in the inode operation table of the `proc` virtual file system. The replaced function would not return the ID of the hidden processes, so they would neither appear in the `/proc` directory nor in the result of `ps` command.

  We implemented the monitor to keep checking the value of `proc_root.proc_iops->lookup` not be modified during the target's execution. The monitor is implemented in a naive way; hard coded to read the fixed pointers. However in addition, as presented in SBCFI [50] and OSck [33] papers, these constant pointers can be extracted automatically by analyzing the source code and the on-line memory image of the target.

- **hideme** and **pidmap-hideme** rootkits modifies the data structures referenced when listing the entries in the `/proc` directory[33]. The process IDs of running processes are managed with two kernel data structures: `pidhash` and `pidmap`. `hideme` removes an entry from `pidhash` so the corresponding process ID would not appear in the `/proc` directory. `pidmap-hideme` clears a bit that represents the use of corresponding process ID, so searching IDs would skip the number as if is not allocated by any process. Hiding its ID, malicious process can remain in the task list while not listed in the `/proc` directory. Our monitor detects the hidden process by traversing all the entries in the task list, checks each task if its ID can be found in `pidhash` and `pidmap`.

- **swipemem** is a rootkit that tries to fill out the xv6 kernel's memory area with zeros. It runs as a Linux kernel module, modifies page tables to map the target memory with a write permission; Linux does not map physical address areas that are outside its allocated memory. We developed `swipemem` by ourselves to show the secure pager's ability to detect malicious memory update from untrusted target.

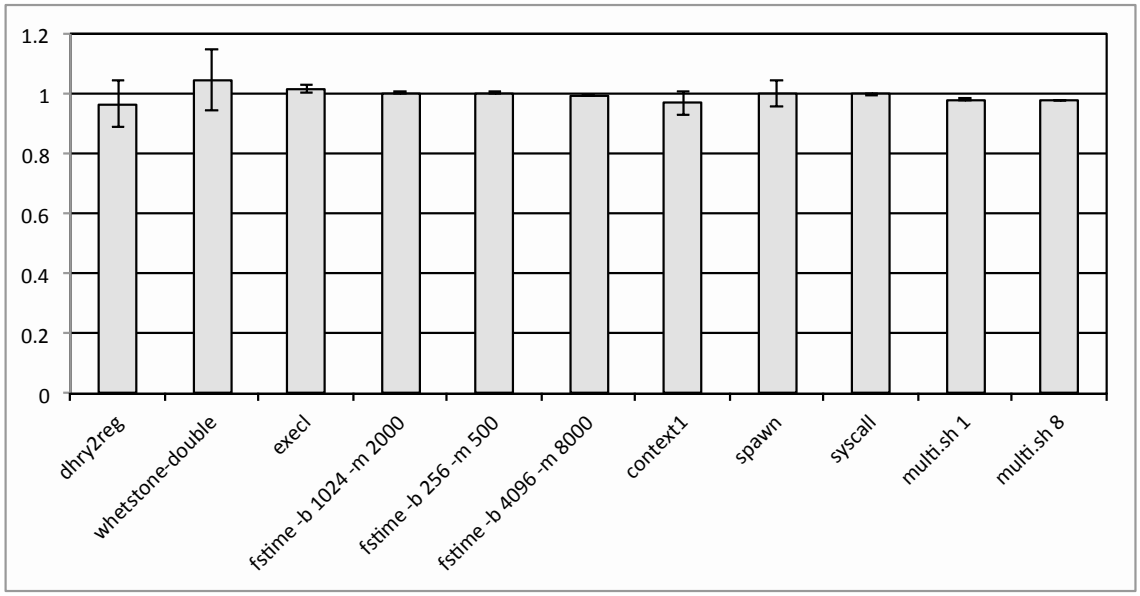


Figure 5.9: The result of Unixbench on Linux with 3 cores running beside the monitor and the secure pager. Normalized to the score of native Linux with 3 cores.

5.5.2 Performance

One of our interests is to estimate the effect of the secure pager to the performance of the target system. In this section we evaluate how the monitor affects the performance of applications running on the target, Linux. The evaluations were executed under QEMU's simulated environment running on Dell Optiplex 755 with Intel Core 2 Quad Q6600 2.4GHz CPU, 2GB main memory. As the target OS we used Linux 2.6.32 which is modified to run cooperatively with the secure pager. The secure pager based on the xv6 kernel revision 4 [22]. The monitor runs as an application of xv6.

Since the monitor runs statically on top of a dedicated core and assigned its own memory region, it would not directly affect the execution of Linux. The monitor periodically reads Linux's data structures that may cause contention on accessing the shared main memory. Figure 5.9 shows the result of the Unixbench benchmark suite [] running on Linux with 3 cores running beside the monitor protected by the secure paging. The score is normalized to the score of Linux with 3 cores without the monitor and the secure pager. The performance is not affected in the most cases, where the worst case 3 % per slowdown. A few results show slight speedup which is entailed by virtual environment hosted of QEMU.

Figure 5.10 shows the result of Unixbench on Linux with the secure pager using 3 cores, compared with native Linux running on 4 cores. The maximum computation power assigned to the Linux kernel is $(N-1)/N$ where N is the number of the cores. In this experiment the processor power available for Linux is reduced to Consequently the score is illustrates the performance degradation around 75% the score when Linux is dominating

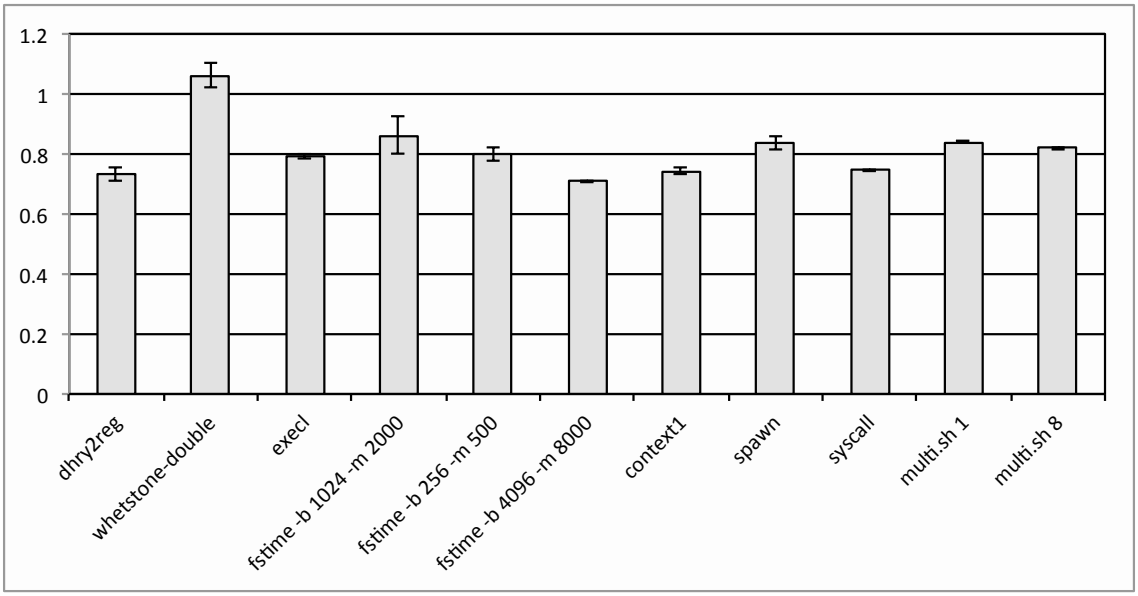


Figure 5.10: The result of Unixbench on Linux with 3 cores running beside the monitor and the secure pager. Normalized to the score of native Linux with 4 cores.

the processor. The result of whetstone experienced speedup, which is the limitation of the benchmark. We configured Unixbench to fork benchmarks into 4 processes. The execution of 4 whetstone processes were serialized on the 3 core configuration but did not calculate the increased execution time.

5.5.3 Engineering Cost

Table 5.1 shows the amount of modification we applied to the xv6 kernel to support the method of the secure paging and the ability to load the target OS.

Table 5.2 shows the amount of modification required to run Linux under the LLM architecture.

Table 5.1: Lines of code modified in xv6 (rev4) to create the monitor OS.

Entity	**LOC**
Unmodified xv6 (revision 4)	8688
Changes to xv6	404
Secure pager	592
SHA1 (from RFC 3174)	539
OS loader (loads target)	146
Total modifications	1681

Table 5.2: Lines of code modified in Linux to run on the LLM architecture.

Entity	**LOC**
Linux 2.6.32	1431

5.6 Related Work

Hypervisors have been widely used as a research vehicle for monitoring the integrity of OS. Because modern OS kernels became increasingly large in their sizes, they are one of the primary targets of malware. As the size of the kernel increases, the possibility of making security holes increases. Even if an intrusion detection system (IDS) has no vulnerabilities, malware may exploit an untrustworthy OS kernel and indirectly disable its detection. For instance, a rootkit modifies kernel data structures to make itself not appear in the list of processes but still scheduled, so that IDS will not investigate it. The above problem motivated some researches to run IDS *outside* an OS with help of hypervisors [25]. The monitoring service is handled within a hypervisor or within another OS that has privilege to access the state of the target OS.

There are a number of researches on monitoring the target kernel. Patagonix ensures the execution of binaries that appear in the list in the target [42]. In this chapter we adopted the method of tracking the control-flow integrity of the kernel. SBCFI [50] and Osck [33] extracts the data structures of Linux kernel and traverse them at runtime. Our case studies detected malware using the approach based on these researches.

Some researches claim hypervisors cannot be treated as the root of trust. As hypervisors enhanced their functionalities, like the OSes in the past, their sizes have also increased. Thus there are some researches on making hypervisors trustworthy. Hypersafe has enhanced the security by enforcing $W \bigoplus X$ to the memory of the hypervisor to prevent rewriting its code [63]. seL4 applied formal verification to a microkernel to prove it is free of bugs [38]. However their implementation does not meat real requirement. In order to prevent the state explosion, seL4 does not support interrupts; instead it handles signals by polling. In addition it is not applicable to the existing hypervisors written in C/C++. NOVA proposed a minimal implementation of a hypervisor [56]. Making the root of trust small may secure the overall system, however we still need studies on balancing performance, applicability and security of these methods.

Overshadow encrypts the pages of the virtualized target so they cannot be obtained by malware [21]. The plain page is managed by the underlying hypervisor and mapped to the process that has privilege to access the secret. Overshadow identify the running process with help of the hypervisor, if the other process tries the access the hidden pages, they are remapped to encrypted pages. Their method is similar to the secure pager at the point of providing an illusion of accessing memory pages but in real they are swapped by

the manager.

The idea of the LLM architecture and the secure paging is more close to the idea of architectural approaches. Gibraltar [15] leverages a PCI card connected to the target machine. Monitor runs on an independent machine which obtain the memory pages the target through the network and monitors its invariants. Alike to our method, the monitor is isolated from the target. XOM is a customized processor which has a capability of executing encrypted code [43]. It leverage asymmetric key cryptosystem to encrypt the code stored in the main memory. It also uses encryption hash to detect the modification of data. XOM encrypted a module running in the same address space as an OS kernel, whose integrity is guaranteed but the kernel can alter the module to disable its functionality. Cell Broadband Engine [53] supports executing an encrypted secret loaded in the core-local memory of its co-processors (SPE). When SPE is executing an encrypted code, the other SPE and the main control processor cannot access the local memory. This is similar to the idea of the LLM architecture. We believe our method can be implemented to existing processors with minor changes, which requires less effort on developing specialized architecture.

5.7 Summary

In this chapter we proposed the LLM architecture that provides isolated execution environment using core-local memory for securely monitoring the target kernel. Furthermore proposed the secure paging which enables virtually extending the size of core-local memory, a method to store the memory pages of the monitor in the shared main memory with keeping their integrity. The case study in emulated environment succeeded to detect some real security risks in the target and also the attack to the monitor itself.

Chapter 6

Conclusion

Modern mobile devices have evolved to host multiple OSes in a single device. While various research efforts have shown the advantages of using virtualization in mobile devices, the role of hypervisors in embedded systems are getting more general as those in desktop/enterprise systems. This direction makes the design of a multi-OS embedded system more complex. In this dissertation we take the position that some applications in embedded system virtualization can be supported with more simple method with help of minor assumptions on the processor architecture.

We first showed the real-time responsiveness of the virtualized RTOS is interfered by the underlying hypervisor. Focusing on the real-time scheduling of RTOS and GPOS running on a processor, we proposed a light-weight virtualization layer SPUMONE which runs on real-world embedded processor. SPUMONE achieved better real-time responsiveness than the previous approach with less engineering effort. In addition the interrupt priority level assignment and the core-migration reduced the interrupt delay of RTOS entailed by the activities of GPOS.

Next we proposed a method to host a security monitor without help of pure-hypervisor layer. The method assigns a *privileged core* to the monitor, instead of *privileged level*, that provide an isolation immutable by means of software. We implemented the proof of the concept on a machine emulator, succeeded to detect malicious applications in the target OS and also the attack to the monitor itself.

Bibliography

[1] Iguana. URL http://www.ertos.nicta.com.au/software/kenge/iguana-project/latest/.

[2] L4-Embedded. URL http://www.ertos.nicta.com.au/research/l4/embedded.pml.

[3] National Vulnerability Database. URL http://nvd.nist.gov/.

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

[5] Packet Storm. URL http://packetstormsecurity.org/.

[6] PDAdb.net. URL http://pdadb.net/.

[7] SH-Mobile Series. URL http://www.renesas.com/products/mpumcu/sh_mobile/sh_mobile_landing.jsp.

[8] Mobile Virtualization. URL http://www.redbend.com/.

[9] Mobile Handset Design: Realizing Flexible, Low Cost, Higher Security, Device Management Using OpenOS and Real-Time Virtualization. Technical report, 2006.

[10] The SCC Programmer's Guide revision 0.61 - Intel Research, 2010. URL techresearch.intel.com/spaw2/uploads/files/SCCProgrammersGuide.pdf.

[11] The SCC Platform Overview - Intel Research, 2010. URL techresearch.intel.com/spaw2/uploads/files/SCC_Platform_Overview.pdf.

[12] Single Chip Clould Computer: An experimental many-core processor from Intel Labs, Mar. 2010. URL communities.intel.com/servlet/JiveServlet/downloadBody/5075-102-1-8132/SCC_Sympossium_Mar162010_GML_final.pdf.

[13] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole. A measurement-based analysis of the real-time performance of linux. In *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, pages 133–142, 2002.

[14] F. Armand and M. Gien. A Practical Look at Micro-Kernels and Virtual Machine Monitors. In *Proceedings of the 6th IEEE Consumer Communications and Networking Conference (CCNC'09)*, pages 1–7, 2009.

[15] A. Baliga, V. Ganapathy, and L. Iftode. Detecting Kernel-level Rootkits using Data Structure Invariants. *IEEE Transactions on Dependable and Secure Computing*, 8(4), July 2011.

[16] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM symposium on Operating systems principles (SOSP'03)*, pages 164–177. New York, NY, USA, ACM Request Permissions, 2003.

[17] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. The VMware mobile virtualization platform. *ACM SIGOPS Operating Systems Review*, 44(4):124, Dec. 2010.

[18] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: a New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09)*. ACM Request Permissions, Oct. 2009.

[19] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2005.

[20] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*, pages 43–57. USENIX Association, Dec. 2008.

[21] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008)*. Seattle, WA, USA, Mar. 2008.

[22] R. Cox, M. F. Kaashoek, and R. T. Morris. Xv6, a simple Unix-like teaching operating system. Technical report.

[23] K. Duda and D. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. *ACM SIGOPS Operating Systems Review*, 33(5):261–276, 1999.

[24] FSMLabs. RTLinux. URL http://www.fsmlabs.com/.

[25] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. *In Proceedings of the Internet Society's 2003 Symposium Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[26] R. Goldberg. ARCHITECTURE OF VIRTUAL MACHINES. In *Proceedings of the Workshop on Virtual Computer Systems*. ACM, Mar. 1973.

[27] R. Goldberg. Survey of virtual machine research. *IEEE Computer*, 1974.

[28] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an Application Program. In *USENIX 1990 Summer Conference*, page 8795, 1990.

[29] H. Härtig, M. Hohmuth, J. Liedtke, and S. Schönberg. The Performance of Micro-Kernel-Based Systems. *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 66–77, 1997.

[30] H. Härtig, M. Hohmuth, and J. Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART'98)*, 1998.

[31] G. Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems (IIES'08)*, pages 11–16. New York, NY, USA, ACM, 2008.

[32] G. Heiser and B. Leslie. The OKL4 microvisor: convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems (APSys 2010)*, pages 19–24, New York, New York, USA, 2010. ACM Press.

[33] O. Hofmann, A. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with OSck. In *Proceedings of the 16th international conference on Architectural support for programming languages and operating systems (ASPLOS'11)*, Mar. 2011.

[34] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim. Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones. In *Proceedings of the 5th IEEE Consumer Communications and Networking Conference (CCNC'08)*, pages 257–261, 2008.

[35] Y. Ishiwata. ART-Linux 2.6 for Single CPU: Design and Implementation. In *Proceedings of Digital Human Symposium 2009*, Mar. 2009.

[36] Y. Ishiwata and T. Matsui. Development of Linux which has advanced real-time processing function. In *Proceedings of 16th Annual Conference of Robotics Society of Japan*, pages 355–356, 1998.

[37] M. Ito, T. Hattori, Y. Yoshida, K. Hayase, T. Hayashi, O. Nishii, Y. Yasu, A. Hasegawa, M. Takada, H. Mizuno, K. Uchiyama, T. Odaka, J. Shirako, M. Mase,

K. Kimura, and H. Kasahara. An 8640 MIPS SoC with Independent Power-Off Control of 8 CPUs and 8 RAMs by An Automatic Parallelizing Compiler. *Digest of Technical Papers in IEEE International Solid-State Circuits Conference, 2008 (ISSCC 2008)*, pages 90–598, 2008.

[38] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220. New York, NY, USA, ACM Request Permissions, 2009.

[39] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization andaverage case behavior. In *Proceedings of Real-Time Systems Symposium*, pages 166–171, 1989.

[40] B. Leslie, C. van Schaik, and G. Heiser. Wombat: A portable user-mode Linux for embedded systems. In *Proceedings of the 6th Linux Conference*, 2005.

[41] J. LeVasseur, V. Uhlig, and Y. Yang. Pre-virtualization: soft layering for virtual machines. In *Proceedings of the 13th Asia-Pacific Computer Systems Architecture Conference, 2008 (ACSAC'08)*, 2006.

[42] D. Lie and L. Litty. Using hypervisors to secure commodity operating systems. In *Proceedings of the Fifth ACM Workshop on Scalable Trusted Computing (STC '10)*, Oct. 2010.

[43] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In *Proceedings of the 19 ACM symposium on Operating systems principles (SOSP '03)*, pages 178–192, 2003.

[44] P. Mantegazza, E. Dozio, and S. Papacharalambous. RTAI: Real Time Application Interface. *Linux Journal*, 2000(72es), Apr. 2000.

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

[46] montavista. MontaVista Linux. URL http://www.mvista.com/.

[47] T. Nakajima, T. Kitayama, H. Arakawa, and H. Tokuda. Integrated management of priority inversion in Real-Time Mach. In *Proceedings of Real-Time Systems Symposium, 1993*, pages 120–130, 1993.

[48] S. Oikawa and M. Ito. Experiences of Building Linux/RTOS Hybrid Operating Environments on Virtual Machine Monitors. *IJCSNS*, 6(5A):146, 2006.

[49] S. Oikawa, H. Ishikawa, M. Iwasaki, and T. Nakajima. Constructing secure operating environments by co-locating multiple embedded operating systems. In *Proceedings of the Second IEEE Consumer Communications and Networking Conference (CCNC'05)*, pages 43–48, 2005.

[50] N. Petroni, Jr, and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS '07)*, Oct. 2007.

[51] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[52] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th conference on USENIX Security Symposium (SSYM'00)*, 2000.

[53] K. Shimizu, S. Nusser, W. Plouffe, V. Zbarsky, M. Sakamoto, and M. Murase. Cell Broadband Engine processor vault security architecture. *IBM Journal of Research*, 2010.

[54] T. Shimosawa, H. Matsuba, and Y. Ishikawa. Logical Partitioning without Architectural Supports. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications, 2008 (COMPSAC '08)*, pages 355–364, 2008.

[55] V. Skoglund. Towards Scalable Multiprocessor Virtual Machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, 2004.

[56] U. Steinberg and B. Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European conference on Computer systems (EuroSys 2010)*, pages 209–222, New York, New York, USA, 2010. ACM Press.

[57] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[58] System Architecture Group. L4Ka::Pistachio microkernel. URL http://l4ka.org/projects/pistachio/.

[59] H. Takada, S. Iiyama, T. Kindaichi, and S. Hachiya. Linux on ITRON: a hybrid operating system architecture for embedded systems. In *Proceedings of the Symposium on Applications and the Internet (SAINT'02)*, pages 4–7, 2002.

[60] TOPPERS Project. TOPPERS. URL http://www.toppers.jp/.

[61] L. Torvalds et al. The Linux Kernel. URL http://www.kernel.org/.

[62] P. Varanasi and G. Heiser. Hardware-supported virtualization on ARM. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*.

[63] Z. Wang and X. Jang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[64] P. M. Wells, K. Chakraborty, and G. S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques (PACT '06)*. ACM Request Permissions, Sept. 2006.

[65] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the USENIX Annual Technical Conference 2002*, 2002.

[66] V. Yodaiken. The RTLinux Manifesto. In *Proceedings of the 5th Linux Expo*, 1999.

[67] Y. Yoshida, T. Kamei, K. Hayase, S. Shibahara, O. Nishii, T. Hattori, and Hasegawa. A 4320MIPS Four-Processor Core SMP/AMP with Individually Managed Clock Frequency for Low Power Consumption. *Digest of Technical Papers in IEEE International Solid-State Circuits Conference, 2007 (ISSCC 2007)*, pages 100–590, 2007.

Publication List

[1] Yuki Kinebuchi, Hitoshi Mitake, Yohei Yasukawa, Takushi Morita, Alexandre Courbot, and Tatsuo Nakajima. A Study on Real-time Responsiveness on Virtualization based Multi-OS Embedded Systems. In César Benavente-Peces and Joaquim Filipe, editors, *Proceedings of the 1st International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS 2011)*, pages 369–378. SciTePress, 2011.

[2] 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 the 13th Annual Linux Symposium (OLS 2011)*, pages 241–249, Ottawa, Canada, June 2011.

[3] Hiromasa Shimada, Alexandre Courbot, Yuki Kinebuchi, and Tatsuo Nakajima. A Software Infrastructure for Dependable Embedded Systems. *International Journal of Computer Systems Science and Engineering (IJCSSE), Special Issue on Real-time Systems*, 2011.

[4] Hiromasa Shimada, Yuki Kinebuchi, Tsung-Han Lin, Alexandre Courbot, and Tatsuo Nakajima. Design issues in composition kernels for highly functional embedded systems. In *SAC '11: Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM Request Permissions, March 2011.

[5] Tatsuo Nakajima, Yuki Kinebuchi, Hiromasa Shimada, Alexandre Courbot, and Tsung-Han Lin. Temporal and spatial isolation in a virtualization layer for multi-core processor based information appliances. *ASPDAC '11: Proceedings of the 16th Asia and South Pacific Design Automation Conferenc*, pages 645–652, 2011.

[6] Tsung-Han Lin, Yuki Kinebuchi, Hiromasa Shimada, Hitoshi Mitake, Chen-Yi Lee, and Tatsuo Nakajima. Hardware-Assisted Reliability Enhancement for Embedded Multi-core Virtualization Design. In *RTCSA '11: Proceedings of the 2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE Computer Society, August 2011.

[7] Ning Li, Yuki Kinebuchi, and Tatsuo Nakajima. Enhancing Security of Embedded Linux on a Multi-core Processor. In *RTCSA '11: Proceedings of the 2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE Computer Society, August 2011.

[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 Multi-core Virtualization Design. In *ISORC '11: Proceedings of the 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE Computer Society, March 2011.

[9] Hitoshi Mitake, Yuki Kinebuchi, Alexandre Courbot, and Tatsuo Nakajima. Coexisting real-time OS and general purpose OS on an embedded virtualization layer for a multicore processor. In *SAC '11: Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM Request Permissions, March 2011.

[10] Tatsuo Nakajima, Yuki Kinebuchi, Alexandre Courbot, Hiromasa Shimada, Tsung-Han Lin, and Hitoshi Mitake. Composition Kernel: A Multi-core Processor Virtualization Layer for Highly Functional Embedded Systems. In *Proceedings of IEEE 16th Pacific Rim International Symposium on Dependable Computing (PRDC 2010)*, December 2010.

[11] Hiromasa Shimada, Alexandre Courbot, Yuki Kinebuchi, and Tatsuo Nakajima. A Lightweight Monitoring Service for Multi-core Embedded Systems. In *ISORC '10: Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE Computer Society, May 2010.

[12] Yuki Kinebuchi, Tatsuo Nakajima, Vinod Ganapathy, and Liviu Iftode. Core-Local Memory Assisted Protection. In *PRDC '10: Proceedings of the 2010 IEEE 16th Pacific Rim International Symposium on Dependable Computing*. IEEE Computer Society, December 2010.

[13] Yuki Kinebuchi, Kazuo Makijima, Takushi Morita, Alexandre Courbot, and Tatsuo Nakajima. Composition kernel: a software solution for constructing a multi-OS embedded system. *EURASIP Journal on Embedded Systems*, 2010, January 2010.

[14] Hitoshi Mitake, Yuki Kinebuchi, Alexandre Courbot, and Tatsuo Nakajima. Handling Lock-Holder Preemption in Real-Time Virtualization Layer for Multicore Processors. In *Proceedings of Work-In-Progress Session of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2010)*, 2010.

[15] Tatsuo Nakajima, Yuki Kinebuchi, Alexandre Courbot, Hiromasa Shimada, Tsung-Han Lin, and Hitoshi Mitake. Composition Kernel: A Multi-core Processor Virtualization Layer for Highly Functional Embedded Systems. In *PRDC '10: Proceedings of the 2010 IEEE 16th Pacific Rim International Symposium on Dependable Computing*. IEEE Computer Society, December 2010.

[16] Tatsuo Nakajima, Yuki Kinebuchi, Alexandre Courbot, Hiromasa Shimada, Tsung-Han Lin, and Hitoshi Mitake. Composition kernel: a multi-core processor virtualization layer for rich functional smart products. In *SEUS'10: Proceedings of the 8th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*. Springer-Verlag, October 2010.

[17] Yuki Kinebuchi, Takushi Morita, Kazuo Makijima, Midori Sugaya, and Tatsuo Nakajima. Constructing a Multi-OS Platform with Minimal Engineering. In Achim Rettberg, Mauro C Zanella, Michael Amann, Michael Keckeisen, and Franz J Rammig,

editors, *IFIP Advances in Information and Communication Technology 2009*, pages 195–206, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[18] Yuki Kinebuchi, Wataru Kanda, Yu Yumura, Kazuo Makijima, and Tatsuo Nakajima. A Hardware Abstraction Layer for Integrating Real-Time and General-Purpose with Minimal Kernel Modification. In *STFSSD '09: Proceedings of the 2009 Software Technologies for Future Dependable Distributed Systems*. IEEE Computer Society, March 2009.

[19] Lei Sun, Yuki Kinebuchi, Tomohiro Katori, and Tatsuo Nakajima. Runtime Self-Diagnosis and Self-Recovery Infrastructure for Embedded Systems. In *SASO '09: Proceedings of the 2009 Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE Computer Society, September 2009.

[20] Yutaka Ishikawa, Hajime Fujita, Toshiyuki Maeda, Motohiko Matsuda, Midori Sugaya, Mitsuhisa Sato, Toshihiro Hanawa, Shinichi Miura, Taisuke Boku, Yuki Kinebuchi, Lei Sun, Tatsuo Nakajima, Jin Nakazawa, and Hideyuki Tokuda. Towards an Open Dependable Operating System. In *ISORC '09: Proceedings of the 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE Computer Society, March 2009.

[21] Tatsuo Nakajima, Hiroo Ishikawa, Yuki Kinebuchi, Midori Sugaya, Sun Lei, Alexandre Courbot, Andrej Zee, Aleksi Aalto, and Kwon Ki Duk. An Operating System Architecture for Future Information Appliances. In *SEUS '08: Proceedings of the 6th IFIP WG 10.2 international workshop on Software Technologies for Embedded and Ubiquitous Systems*. Springer-Verlag, October 2008.

[22] Yuki Kinebuchi, Midori Sugaya, Shuichi Oikawa, and Tatsuo Nakajima. Task Grain Scheduling for Hypervisor-Based Embedded System. In *HPCC '08: Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*. IEEE Computer Society, September 2008.

[23] Wataru Kanda, Yu Yumura, Yuki Kinebuchi, Kazuo Makijima, and Tatsuo Nakajima. SPUMONE: Lightweight CPU Virtualization Layer for Embedded Systems. In *EUC '08: Proceedings of the 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*. IEEE Computer Society, December 2008.

[24] M Sugaya, Y Kinebuchi, and S Oikawa. VPE: Virtual Periodic Execution for Embedded System. *Proceedings of the Work-in-Progress Session: the 18th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2007.

[25] Yuki Kinebuchi, Hidenari Koshimae, and Tatsuo Nakajima. Constructing machine emulator on portable microkernel. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*. ACM Request Permissions, March 2007.

[26] Yuki Kinebuchi, Hidenari Koshimae, Shuichi Oikawa, and Tatsuo Nakajima. Dynamic translator-based virtualization. In *SEUS'07: Proceedings of the 5th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*. Springer-Verlag, May 2007.

[27] H Koshimae and Y Kinebuchi. Using a processor emulator on a microkernel-based operating system. *Proceedings of the Work-in-Progress Session: the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2006.

[28] Y Kinebuchi, H Koshimae, and S Oikawa. Virtualization techniques for embedded systems. *Proceedings of the Work-in-Progress Session: the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2006.