Experiences of Building Linux/RTOS Hybrid Operating Environments on Virtual Machine Monitors

Shuichi Oikawa,[†] *and Megumi Ito*[††]*,*

University of Tsukuba, Ibaraki, JAPAN

Summary
This paper presents our experiences of building Linux/RTOS hybrid operating environments on Xen and Gandalf virtual machine monitors (VMMs). Xen is a popular open source VMM while Gandalf is our in-house virtual machine monitor that was designed and implemented from scratch to be a simple yet extremely lightweight VMM. We ported an RTOS to both Xen and Gandalf, which were enabled to host multiple RTOSes along with Linux. One significant advantage of employing a VMM to construct such a hybrid environment is that OSes executed on a VMM can be spatially and temporally protected from each other. Our experiences and evaluations show that Gandalf's approach, which combines full- and para-virtualization methods, has clear advantages in terms of both implementation cost and runtime cost.

Key words:
Virtual Machine Monitors, Operating Systems, Real-Time Systems, Information Appliances.

Introduction

The current embedded systems, especially consumer electronics products, involve a number of complicated requirements for their operating environments that are difficult to satisfy them together at once by a single operating system (OS). In order to cope with conflicting requirements imposed by such complex embedded systems, the hybrids of a general purpose OS and a real-time operating system (RTOS) have been developed. While a general purpose OS is good at the provision of GUI with its rich set of middleware support, an RTOS covers real-time processing. Typically in those hybrids, a general purpose OS kernel and an RTOS's application tasks share the same most privileged level of a processor without protection. This hybrid model ay work well if a system is designed from scratch having applications properly classified into real-time and non real-time tasks. In reality, however, the existing software resources inherited from the past products need to be reused; thus, undesired programs are also brought into the most privileged level to run on an RTOS, and tend to become sources of problems because of lack of protection.

This paper presents our experiences of building Linux/RTOS hybrid operating environments on Xen and Gandalf virtual machine monitors (VMMs). Xen is a popular open source virtual machine monitor while Gandalf is our in-house virtual machine monitor that was designed and implemented from scratch to be a simple yet extremely lightweight VMM combining full- and para-virtualization methods. As a general purpose OS and an RTOS hosted on VMMs, we use Linux and µITRON [15], respectively. The implementations of those hybrid operating environments are on the PC/AT compatible platform with the Intel IA-32 processor. We chose µITRON and Linux because of their popularity. In Japan µITRON is the RTOS that has been the most widely used in a variety of products, so that industries have a huge amount of the existing software resources. Linux's popularity is recently increasing for larger embedded systems. Unlike the existing approach described above, only VMMs execute at the most privileged level, and have them execute within their own isolated protection domains; thus, hosted OSes can be spatially and temporally protected from each other. Additionally, multiple RTOS instances can be hosted along with a general purpose OS.

We describe our experiences of porting an RTOS and Linux to Xen and Gandalf, and show from evaluations that Gandalf's approach, which combines full- and para-virtualization methods, has clear advantages in terms of both implementation cost and runtime cost.

Background and Related Work

Making hybrids of an RTOS and a general purpose OS is not a new idea. As a practical approach to deal with complex systems, several of them have been developed, and some are widely used. [3] introduced the executive that support the co-residence of an RTOS and a general purpose OS. RTLinux [1] and RTAI [8], which are poplar among Linux users, have Linux kernel execute on an RTOS kernel. There are some commercial products, such as Accel-Linux [1] and Linux on NORTi, [2] that enable µITRON to run aside of Linux kernel.

All of them take the same approach, which is that an RTOS kernel, RTOS's application tasks, and a general

Manuscript received May 5, 2006.
Manuscript revised May 25, 2006.

[1] http://www.elwsc.co.jp/english/products/accel_linux.html
[2] http://www.embedded-sys.co.jp/bios/index.htm

purpose OS kernel share the same protection domain at the most privileged level; thus, there is noprotection among them and hardware. Such lack of protection may not be a problem if a system is designed from scratch having applications properly classified into real-time and non real-time tasks. After proper classification, there should be only a small number of RTOS's application tasks that specifically require real-time execution. It is, however, problematic if the existing applications on an RTOS are simply reused by taking advantage of the hybrid of an RTOS and a general purpose OS. In such a system, there tend to be a larger number of RTOS's application tasks, which are brought from the past products. Since RTOS's application tasks run at the most privileged level within the same protection domain as RTOS and general purpose OS kernels, their misbehavior is directly connected to system malfunction or a crash. A general purpose OS kernel also can be a source of problems because of its execution at the most privileged level. A general purpose OS kernel for the hybrid with an RTOS is usually modified not to touch hardware's interrupt controlling functions, so that interrupts are not disabled for a indeterministically long time. Since a general purpose OS kernel is still allowed to disable interrupts by controlling hardware, there are chances to introduce kernel modules that are not properly modified for the hybrid; thus, they cause temporal malfunction.

Our approach is that a VMM, such as Xen and Gandalf, hosts RTOSes and a general purpose OS. This approach enables the provision of spatial and temporal protection, which is missing in the existing hybrid systems. With our approach, spatial protection is implemented by having OSes run within their own protection domains. Since there is no means provided to corrupt or steal the programs or data of the other OSes, OSes' misbehavior does not affect the execution of the other OSes. Temporal protection is realized by limiting hardware access by OSes. The OSes hosted on Gandalf execute at a less privileged level, at which only limited hardware access is permitted; thus, the hosted OSes cannot directly disable interrupts at hardware's interrupt controller. Therefore, temporal malfunction incurred by disabling interrupts can be avoided.

The development of VMMs has a long history beginning with IBM CP/67 [10] followed by IBM VM/370 [5] and its successors for mainframe computers. Since a few years ago VMMs revived to be a hot research and development topic because of VMMs' capability to accommodate multiple operating systems in a single system. Researchers and developers consider VMMs an excellent software tool to deal with increasing reliability and security. In order to achieve better performance on commodity platforms, which cannot be virtualized efficiently [12], para-virtualization was introduced [2,13,16]. With para-virtualization, VMM developers define easily and efficiently virtualizable hardware as interface for OSes to communicate with a VMM. Para-virtualization can be used for the whole system platforms [2] or partially for I/O devices only [14]. In contrast to para-virtualization, which defines non-existing virtual hardware, the method first taken by mainframe VMMs is called full-virtualization, which defines the same interface as the existing real hardware.

Xen takes para-virtualization method while Gandalf VMM takes the hybrid of para- and full-virtualization methods. Gandalf exports a virtual processor interface for RTOSes while it enables a general purpose OS to run on it with limited modifications. Gandalf's hybrid virtualization method can balance implementation cost and runtime cost.

Paper Organization

The rest of this paper is organized as follows. The next section describes our system building experiences of the Linux/RTOS hybrid operating environments on the two VMMs, Xen and Gandalf, along with their system overviews. Section 3 evaluates and compares the two hybrid operating environments quantitatively and qualitatively. Finally, Section 4 summarizes the paper.

2. Hybrid Operating Environments

This section describes our system building experiences of the Linux/RTOS hybrid operating environments on Xen and Gandalf. For each of them, we describe the overview of a VMM, on which a hybrid operating environment is built, and the work required to port OSes on the VMM.

Fig. 1 shows the overall architecture of a Linux/RTOS hybrid operating environment we built upon a VMM.

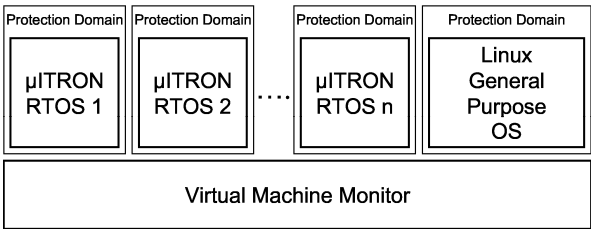

Fig. 1 Overall Architecture of Linux/RTOS Hybrid Operating Environment on VMM

There are multiple OSes running on a VMM. One of OSes is Linux, and the others are RTOSes. RTOSes are

µITRON[3] in our environments. OSes run within their own isolated protection domains; thus, no access from an OS to the other OSes is permitted. A VMM allows hosted OSes only limited hardware access and schedules those OSes to run; thus, those OSes cannot monopolize the whole CPU time. Therefore, OSes hosted on a VMM can be spatially and temporally protected from each other.

2.1 Xen

We first describe the hybrid operating environment on Xen. We first present the overview of Xen, and then describe our experience of porting µITRON. Since Xen is distributed with Linux and NetBSD as its guest OSes, porting Linux on Xen is described elsewhere [2].

2.1.1 Overview

Xen [2] is a VMM based on a virtualization method called *para-virtualization* [2,13,16]. Para-virtualization lets a VMM define its own interface for OSes to control hardware resources. A processor's interface to control hardware resources is privileged instructions, and an OS's machine dependent layer uses those instructions to set up operating environments for user programs. Since para-virtualization alters such interface to control hardware resources, an OS's machine dependent layer needs to be modified to run on Xen. The application binary interface (ABI) remains the same, so that the existing applications require no modification.

Xen employs a hypercall mechanism and shared memory as its para-virtualization interface for guest OSes to communicate with Xen. Hypercalls are analogous to system calls. While system calls are used by user programs to communicate with the OS kernel, hypercalls are used by guest OSes to communicate with the VMM. When a guest OS kernel needs to execute privileged instructions in order to set up and control operating environments for its user programs, the kernel issues hypercalls for that purpose. Along with hypercalls, shared memory is also used as an efficient data transfer channel.

2.1.2 Porting µITRON on Xen

We ported µITRON RTOS on Xen to build a Linux/RTOS hybrid operating environment. Although the source code of µITRON is very simple, porting µITRON on Xen was not actually a very straightforward task mainly because of lack of Xen's para-virtualization interface documentation. For example, the executable image of µITRON on Xen needs to be linked to start from `0xc0000000` in order to be successfully loaded into memory and to start running as a Xen's guest OS. Since µITRON does not provide virtual memory, the original µITRON that runs on a bare hardware is linked to start from a much lower address, which is `0x100000`. Obviously, there is no fair reason why a guest OS cannot start from such a lower address if it does not use virtual memory. It is simply a implementation limitation of Xen, of which design did not take account of running small RTOSes without using virtual memory.

We needed to understand the usage of Xen's hypercall and shared memory interface from the source code of Linux on Xen. Bringing the implementation of such para-virtualization interface to µITRON's source code was also cumbersome. The para-virtualization interface is rich and complicated enough to support Linux and NetBSD. Although the most of functions provided by the interface is unnecessary for µITRON, they depend upon each other among the interface; thus, we ended up with bringing the whole para-virtualization interface to µITRON's source code even if only few functions are called from it.

Once we obtained enough understanding of Xen's para-virtualization interface, brought the interface implementation to µITRON's source code, and modified it to use the interface, debugging the ported µITRON on Xen was helped by Xen's ability to dynamically create and destroy a guest OS. Xen starts the first Linux as a privileged OS, which retains the capability of creating and destroying other guest OSes. While debugging the ported µITRON on Xen, the usual steps for debugging, which involve building a linked executable image from the source code, running it, and stopping it, can be done without rebooting a target platform.

2.2 Gandalf

We describe the hybrid operating environment on Gandalf. We first present the overview of Gandalf, and then describe our experiences of porting µITRON and Linux to be hosted on Gandalf.

2.2.1 Overview

Gandalf is a VMM that we designed from scratch as a simple and efficient VMM in order to minimize implementation and runtime costs incurred by virtualization. Gandalf exports a para-virtualized processor interface for RTOSes as Xen does for its guest OSes. Gandalf also enables a general purpose OS to run on it with very limited modifications. Since unmodified OSes do not run on Gandalf, Gandalf does not provide full-virtualization strictly; thus, we call this method *nearly* full-virtualization.

We choose to take such hybrid of virtualization methods, para-virtualization for RTOSes and nearly full-

[3] We use TOPPERS/JSP as our µITRON RTOS. The information on TOPPERS/JSP can be found at http://www.toppers.jp/.

virtualization for a general purpose OS, in order to balance implementation cost and runtime cost. A general purpose OS is huge and very complicated software. It tends to be actively updated in order to incorporate new features and to fix their bugs. Applying para-virtualization to such an OS significantly increases implementation cost. It is also hard to maintain the source code modified for para-virtualization since it needs to keep up with rapid updates. On the other hand, the implementation of RTOSes is considerably simpler than a general purpose OS, and it tends to be stable for a long time because keeping reliability is more important than adding new features; thus, once their source code base is modified for para-virtualization, the amount of its maintenance work is limited. Therefore, its implementation cost is negligible. In fact, runtime cost is more important for RTOSes. In order to have unmodified OSes execute at a less privileged level, full-virtualization emulates a subset of hardware. Such emulation costs at runtime. Para-virtualization can decrease the overheads of emulation; thus, using para-virtualization is desired for RTOSes in terms of both implementation and runtime costs.

2.2.2 RTOS on Gandalf

We chose para-virtualization for µITRON as described above in favor of less virtualization overheads at runtime. Para-virtualization replaces privileged instructions, which can be correctly executed only at the most privileged level, with hypercalls, which are analogous to system calls provided by Gandalf. By taking advantage of the knowledge of µITRON's implementation and having Gandalf tailored to execute a µITRON instance in specific segments, only few hypercalls are actually required in order to bring up µITRON on Gandalf.

The current implementation of Gandalf provides µITRON with only three hypercalls as the replacements of privileged instructions. One replaces `lidt` instruction, which is used to register interrupt handlers. Another one replaces `sti` and `cli` instructions, which enables and disables interrupts, respectively. The last one replaces `hlt` instruction, which halts a processor until an interrupt is asserted. While some other privileged instructions are used, they were removed because tailoring Gandalf to provide an execution environment that matches µITRON's expectation makes them no longer needed.

2.2.3 Linux on Gandalf

We use Linux as a general purpose OS. We first consider full-virtualization to support Linux since Linux kernel is huge and complicated software and is actively updated to incorporate new features and to fix their bugs. Truly full-virtualization, however, costs quite expensive to implement a VMM and to execute an unmodified OS on it. It requires a fully virtualizable processor [11], or it is made possible only in return for the overheads of virtualizing all computing resources. For example, an OS kernel is designed to utilize the whole virtual address space made available by a processor. If a processor is not fully virtualizable as most of the current processors, there is no room in the same virtual address space left for locating a VMM in a way that it is protected from its guest OS kernel and user processes. In this case, a VMM requires another virtual address space, and heavy context switching between a OS kernel and a VMM happens at every time the VMM's intervention is needed. Such intervention includes the emulation of privileged instructions, handling interrupts and exceptions, and so on. There are many other virtualization overheads caused by full-virtualization. Achieving practical performance that matches commercial VMMs, such as VMware [14] and Microsoft VirtualPC, requires many techniques that even include on-the-fly binary translation [13]; thus, the provision of truly full-virtualization was dropped from our choice.

We therefore decided to allow a few straightforward modifications to bring up Linux on Gandalf. We call this method *nearly* full-virtualization. Allowing a few modifications enables the significant reduction of both implementation and runtime costs. For example, by reducing the virtual address range used by Linux, we can create room for Gandalf and RTOSes in the same virtual address space. It removes the necessity to switch virtual address spaces at each time when Gandalf or RTOSes is invoked. Such reduction of the virtual address range can be done only by modifying a single line in a Linux source code file. Thirteen lines in seven files are currently modified to achieve our nearly full-virtualization of Linux on Gandalf.

3. Evaluation

This section quantitatively and qualitatively evaluates and compares the two hybrid operating environments we built and described above. We first present quantitative evaluation results, and then perform qualitative evaluation on µITRON implementations on Xen and Gandalf.

3.1 Quantitative Evaluation

This section shows quantitative evaluation results for the comparison of runtime costs between the two hybrid operating environments on Xen and Gandalf. All measurements reported below were performed on the Dell

Precision 470 Workstation with Intel Xeon 2.8GHz CPU.[4] Hyper-threading was turned off, so that all measurements were performed on a single CPU.

3.1.1 Basic Performance Evaluation

We first measured the basic costs related to running an OS on a VMM. We measured the costs of issuing a hypercall, processing a privileged instruction, and OS switching. The costs of issuing a hypercall and processing a privileged instruction were measured using µITRON. OS switching cost is the time consumed to switch from µITRON to Linux and then back to µITRON, which means that the two times of OS switching are involved. Table 1 shows the measurement results obtained from Xen and Gandalf. We used cycle counts obtained from `rdtsc` instruction for these measurements on both Xen and Gandalf. The all numbers shown were the average costs after repeating 1,000 times. The cost of processing a privileged instruction was measured only for Gandalf since Xen uses only hypercalls to handle requests that are usually handled by privileged instructions.

Table 1: Basic Performance Comparisons

	Xen	Gandalf
Null Hypercall	0.43 µ sec	0.37 µ sec
Ignored Privileged Instruction	N/A	0.56 µ sec
OS Switching Cost (round trip)	1.80 µ sec	1.02 µ sec

The results show that the costs of hypercalls on Xen and Gandalf are very similar. Although handling a hypercall on Gandalf is slightly faster, the difference is negligible if we take account of other runtime overheads, which frequently happen during the execution of programs, including cache misses. Since hypercalls use the processor's software interrupt mechanism, there is relatively small room for software implementations to make difference in this case. More interesting is that how much processing a privileged instruction takes longer than handling a hypercall. Processing a privileged instruction involves more steps than handling a hypercall. It consists of identifying the instruction address that caused an exception, [5] fetching an instruction from the address, decoding the instruction, and emulating it. The measurement was done with `hlt` instruction, which is a simple one byte instruction, and it does not include the emulation cost. In case of processing a longer privileged instruction, it takes longer in order to decode and fetch a emulating instruction and its operands.

The OS switching cost on Gandalf was measured by using a pair of `hlt` instruction in Linux and its replacement hypercall in µITRON. On Xen, the `XEN_yield` hypercall was used in µITRON. We presume that the reason of the smaller OS switching cost on Gandalf is because of its use of segments to accommodate µITRON in its own protection domain. While µITRON and Linux share the same virtual memory address space, they do not share their protection domains by using disjoint segments. Therefore, the OS switching cost on Gandalf does not include the cost of switching a virtual memory address space, and its cost becomes less than that of Xen.

3.1.2 Evaluating RTOS

In order to evaluate runtime environment's aspect of the two hybrid operating environments from RTOS's point of view, we measured the timer interrupt intervals in µITRON. We used cycle counts obtained from `rdtsc` instruction for these measurements, too. Fig. 2 and Fig. 3 show the measurement results on Xen and Gandalf. Please note that because of Xen's limitation of the fixed timer interval rate, the timer interval of µITRON on Xen is 10 milliseconds while on Gandalf it is 1 millisecond.

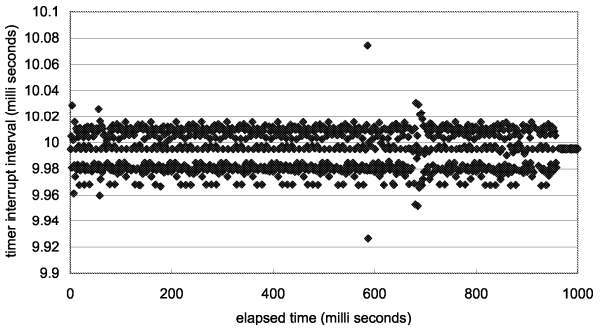

Fig. 2 Timer Interrupt Intervals in µITRON on Xen

[4] Linux reports this CPU as 2794.774 MHz. We use this number to convert cycle counts obtained from `rdtsc` instruction to micro seconds for accuracy.

[5] On an IA-32 processor, the execution of a privileged instruction at a less privileged level causes an exception that is called a general protection fault.

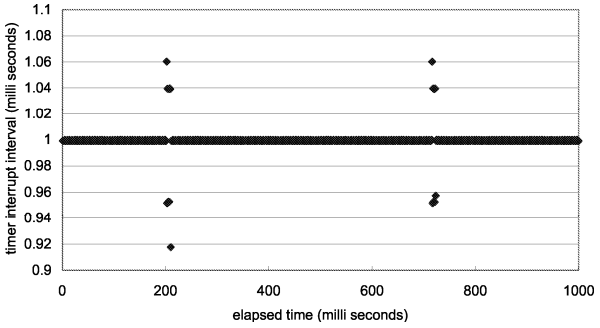

Fig. 3 Timer Interrupt Intervals in µITRON on Gandalf

The measurement results show more jitter is observed on Xen than on Gandalf. Most of the measured timer interrupt intervals on Xen spread in range of 0.04 millisecond (40 µ seconds) between 9.96 and 10.02 milliseconds. In contrast to Xen, on Gandalf the measured timer interrupt intervals are mostly the same at 1 millisecond as the timer device was configured to periodically raise an interrupt every 1 millisecond. There are, however, some spikes around 200 and 700 milliseconds in elapsed time. We need more investigations to be performed in order to find out a cause of these spikes.

Gandalf is apparently more appropriate as a Linux/RTOS hybrid operating environment if those spikes were removed on Gandalf. Since timer interrupts are used to invoke periodic tasks, which are a basis of real-time scheduling, Gandalf's characteristic to provide precise and stable timer interrupts suits with an RTOS.

3.1.3 Evaluating Linux

Finally, in order to evaluate our nearly full-virtualization method used for Linux, we ran several programs included in lmbench benchmark suite [9]. Fig. 4 and Fig. 5 show the results of lmbench programs. We ran the same programs on the original Linux (without virtualization), XenLinux (Dom0), and Gandalf for comparison of performance.

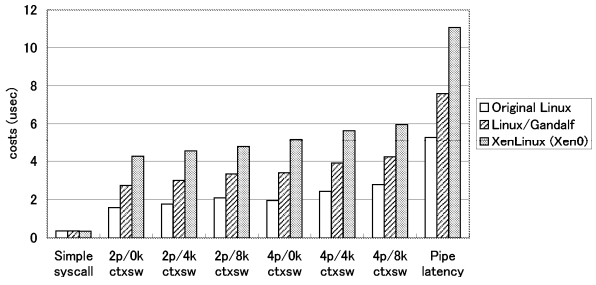

Fig. 4 Linux Performance Comparison (1)

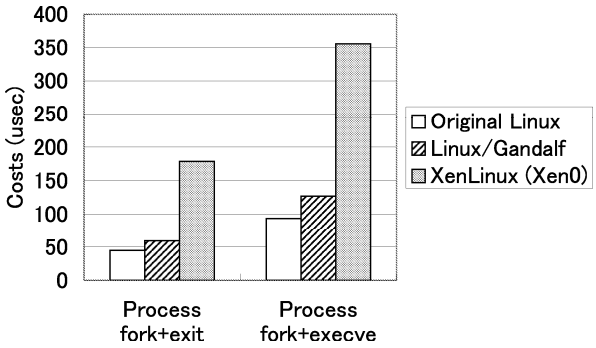

Fig. 5 Linux Performance Comparison (2)

The measurement results show that our nearly full-virtualization method reduces the runtime costs significantly as Linux on Gandalf outperforms XenLinux in all cases. The costs of process fork and exec are even close to the original non-virtualized Linux and significantly better than XenLinux.

3.2 Qualitative Evaluation

As qualitative evaluation, we compare the boot image sizes and the total source code lines of µITRON implementations on Xen and Gandalf. Table 2 shows them along with those of the original µITRON implementation that runs directly on an IA-32 hardware platform. The boot image sizes are the values printed by UNIX size command, which lists the section sizes and shows the total of them. The source code lines include comments and blank lines.

Table 2: Qualitative Comparisons of µITRON Implementations

	on Xen	on Gandalf	Original
Boot Image Size	129KB	87KB	80KB
Total Source Code Lines	20434	12248	13218

Table 2 shows that only µITRON on Xen is significantly larger. Both of the boot image size and the total source code lines are approximately 50% more than the rest of them. This is because µITRON on Xen includes the whole para-virtualization interface for Xen although only few functions are called from µITRON. If we become more familiar with Xen's para-virtualization interface and can design the tailored interface to be used with µITRON on Xen, the boot image size and the total source code lines can be reduced. In this case, we need to maintain the implementation of the tailored interface on our own; thus, it significantly increases the implementation cost. In contrast to Xen, µITRON on Gandalf is more or less the same size as the original one in terms of the boot image size and the total source code lines. We see the reduction of the total source code lines on Gandalf because the hardware interface, especially for

registering interrupt handlers, is simplified. Since only few hypercalls are introduced, we consider the implementation cost of Gandalf on Xen is negligible.

4. Summary

This paper presents our experiences of building Linux/RTOS hybrid operating environments on two VMMs, Xen and Gandalf. Xen is a popular open source VMM while Gandalf is our in-house virtual machine monitor that was designed and implemented from scratch to be tailored to construct a Linux/RTOS hybrid operating environment. Our experiences and evaluations showed that Gandalf's approach, which combines full- and para-virtualization methods, has clear advantages in terms of both implementation cost and runtime cost.

References

[1] M. Barabanov and V. Yodaiken. Real-Time Linux. *Linux Journal*, March 1996.

[2] 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 System Principles*, pp. 164--177, October 2003.

[3] G. Bollella and K. Jeffay. Support for Real-Time Computing within General Purpose Operating Systems - Supporting Co-Resident Operating Systems. In *Proceedings of the 1st IEEE Real-Time Technology and Applications Symposium*, May 1995.

[4] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles*, pp. 143--156, October 1997.

[5] R. J. Creasy. The Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development*, 25 (5), 1981.

[6] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, pp. 34--45, June 1974.

[7] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual.

[8] P. Mantegazza, E. Bianchi, L. Dozio, and S. Papacharalambous. RTAI: Real Time Application Interface. *Linux Journal*, April 2000.

[9] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the USENIX Annual Technical Conference*, pp. 279--294, January 1996.

[10] R. Meyer and L. Seawright. A Virtual Machine Time Sharing System. *IBM Systems Journal*, 9 (3), pp. 199--218, 1970.

[11] G. Popek and R. Goldberg. Formal Requirements for Virtualizable 3rd Generation Architectures. *Communications of the A.C.M.*, 17(7):412--421, 1974.

[12] 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 USENIX Security Symposium*, pp. 129--144, August 2000.

[13] M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Computer*, pp. 39--47, May 2005.

[14] J. Sugerman, G. Venkitachalam, and B. H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of 2001 USENIX Annual Technical Conference*, pp. 1--14, June 2001.

[15] H. Takada ed., μITRON4.0 Specification. TRON Association, 1999. (In Japanese)

[16] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pp. 195--210, December 2002.

Shuichi Oikawa received the B.S., M.S., and Ph.D. degrees in Computer Science from Keio University in 1989, 1991, and 1996, respectively. He is an Associate Professor of the Department of Computer Science at University of Tsukuba from 2004. Before joining University of Tsukuba, he worked at Carnegie Mellon University, Intel Corporation, Sun Microsystems, and Waseda University. His research interests include operating systems, virtual machine monitors, real-time systems, and embedded systems.

Megumi Ito received the B.S. degree in Computer Science from University of Tsukuba in 2006. She is currently a M.S. course student of the Department of Computer Science at University of Tsukuba. Her research interests include operating systems and virtual machine monitors.