Journal of Information Processing Vol. 19 411–420 (Aug. 2011)

Regular Paper

Making a Virtual Machine Monitor Interruptible

Megumi Ito[†1,*1] and Shuichi Oikawa[†2]

This paper describes our approach to making the Gandalf Virtual Machine Monitor (VMM) interruptible. Gandalf is designed to be a lightweight VMM for use in embedded systems. Hardware interrupts are directly notified to its guest operating system (OS) kernel without the interventions of the VMM, and the VMM only processes the exceptions caused by the guest kernel. Since the VMM processes those exceptions with interrupts disabled, the detailed performance analysis using PMC (Performance Monitoring Counters) revealed that the time duration while the interrupts are disabled is rather long. By making Gandalf interruptible, we are able to make VMM based systems more suitable for embedded systems. We analyzed the requirements for making Gandalf interruptible, and designed and implemented mechanisms to achieve this. The experimental results show that making Gandalf interruptible significantly reduces a duration of execution time with interrupts disabled while it does not impact performance.

1. Introduction

As more and more embedded systems are rapidly moving towards having multi-core CPUs in order to balance performance and power consumption, there are increasing needs for virtualized execution environments to be used in those systems. These virtualized execution environments are realized upon virtual machine monitors (VMMs) [4]. VMM based systems enable the provision of secure, reliable, and high functional execution environments.

A major barrier to employing VMMs on embedded systems is their limited resources. In order to overcome such a barrier, we have been developing a lightweight VMM called Gandalf, that targets those resource constrained systems [7),8)]. It currently operates on IA-32 CPUs and two independent Linux operating systems (OSes) concurrently run on it as its guest OSes. The code size and memory footprint of Gandalf is much smaller than that of full virtualization. The number of the modified parts and lines is significantly fewer than in para-virtualization, so that the cost to bring up a guest OS on Gandalf is extremely cheap. Guest Linux on Gandalf performs better than XenLinux. Therefore, Gandalf is an efficient and lightweight VMM that is ideal for resource constrained embedded systems. Those features of Gandalf are made possible by its design. Hardware interrupts are directly notified to its guest OS kernel without the intervention of the VMM, and the VMM only processes exceptions caused by the guest kernel. Since the VMM processes those exceptions with interrupts disabled, a detailed performance analysis using PMC (Performance Monitoring Counters) revealed that a time duration with interrupts disabled is rather long [8)].

This paper describes our effort to make Gandalf interruptible. We analyzed the requirements to make this possible, designed and implemented the mechanisms to achieve this. The experimental results show that making Gandalf interruptible significantly reduces the duration of execution time with interrupts disabled while it does not impact the performance. Our VMM runs only on the IA-32 architecture and does not utilize a hardware assisted virtualization feature, such as Intel VT-x and AMD-V technologies. The importance of the technique described in this paper, however, remains the same. This is because hardware assisted virtualization makes it easier to notify hardware interrupts to guest OS kernels. If VMMs are designed in that way but run with interrupts disabled, systems with those VMMs suffer long interrupt latencies. Therefore, it is important to make VMMs interruptible and to notify hardware interrupts to guest OS kernels as quickly as possible.

The rest of this paper is organized as follows. Section 2 describes the overview of Gandalf VMM. Section 3 proposes a design to made Gandalf interruptible, and Section 4 evaluates performance of interruptible Gandalf. Section 5 describes related work. Finally, Section 6 concludes the paper.

2. Overview of Gandalf

This section first describes the overall architecture of Gandalf, a multi-core CPU oriented lightweight VMM. It targets the IA-32 architecture [6)] as a CPU and Linux as a guest OS. It then shows the preliminary evaluation results that

†1 IBM Research – Tokyo
†2 Department of Computer Science, University of Tsukuba
∗1 Work conducted when she was with University of Tsukuba.

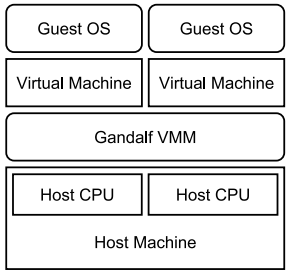


Fig. 1 Structure of Gandalf based system.

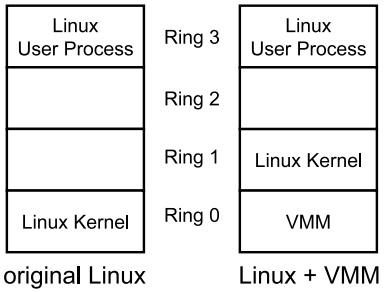


Fig. 2 Privilege level usage.

motivated the work described in this paper.

2.1 Architecture

Gandalf is a Type-I VMM, which is executed directly upon a host physical machine and creates multiple virtual machines for guest OSes. The virtual machines are isolated from each other, so that a guest OS can be executed independently on each virtual machine. **Figure 1** shows the structure of a Gandalf VMM based system. Gandalf keeps the management of physical hardware resources as simple as possible in order to implement a lightweight VMM for embedded systems. Therefore, Gandalf tries to manage resource spatially rather than temporarily whenever possible. For example, Gandalf maps one physical CPU to one virtual CPU while many other VMMs multiplex multiple virtual CPUs on one physical CPU to be shared among multiple virtual machines. Gandalf's spatial resource management scheme enables a simpler and smaller implementation and then leads to a lightweight VMM, while the multiplexing model tends to impose higher overheads for the management of virtual CPUs and virtual machines. In this paper, we use the term VMM interchangeably to mean Gandalf unless otherwise specified.

The IA-32 architecture provides 4 privilege levels (rings) from 0 to 3. The lower the number the higher the privilege level so Ring 0 is the highest privilege level. Some important instructions, which operate on the machine state, are called privileged instructions, and can be executed only in Ring 0. As the left part of **Fig. 2** shows, Linux normally executes its kernel in Ring 0 and its user processes in Ring 3. Thus, the kernel can manage CPUs using privileged instructions and can protect itself from user processes. A VMM needs to be executed in a more privileged (numerically low) level than Linux kernels because the VMM has to manage CPUs and Linux kernels. Therefore, as the right part of Fig. 2 shows, we execute the VMM in Ring 0 and the Linux kernels in Ring 1, which is one level less privileged than the VMM. Because we moved the Linux kernels from Ring 0 to 1, their use of privileged instructions causes general protection faults. The VMM handles those faults to emulate privileged instructions appropriately. The privileged instruction emulator of Gandalf handles faulted instructions. The emulator first reads the instruction words at a faulted address and decodes them to find out which instruction caused the fault. Decoding instructions is complicated especially for IA-32 because of variable length instruction words. A lightweight emulator requires a simpler instruction decoder. Thus, the emulator handles only the privileged instructions that the Linux kernels execute.

Native Linux kernels normally use the entire physical memory in the system. However, when executing multiple Linux kernels on a VMM at the same time, they need to divide up the physical memory. We allocate the upper area of the physical address space for the VMM, divide the remaining area, and allocate a divided part for each Linux. The left most part of **Fig. 3** shows the physical memory map. Shadow paging is used to enforce Linux kernels to use only the allocated physical memories [8]. Shadow paging lets Linux kernels manage their own page tables (guest page tables) and separates them from the shadow page table that is referenced by a physical CPU. The VMM manages the shadow page table in order to keep its consistency with guest page tables and also to observe

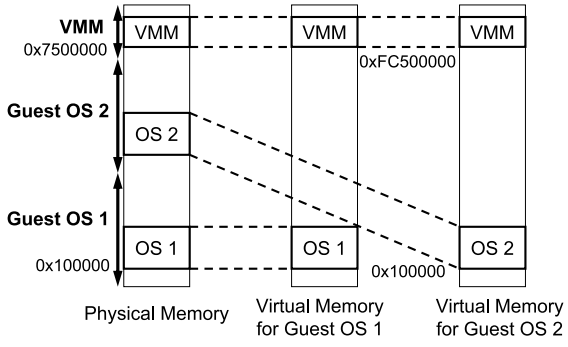


Fig. 3 Memory map.

improper use of physical memories. Concerning a virtual address space, there needs to be an area where a VMM resides. Linux kernels, however, normally use the entire virtual address space, which overlaps the virtual memory area for the VMM. To avoid Linux kernels accessing the VMM, we exclude the virtual memory area for the VMM from the available virtual memory for Linux kernels by modifying the source code ⋆1. We also use the segment mechanism to limit the accessible virtual memory space. For simplicity, we allocate the upper area of the virtual address space to the VMM.

2.2 Interrupt Handling

Embedded systems require quick and timely responses to interrupts. An interrupt can be an event that processes have been waiting for. In that case, the interrupt unblocks those processes. For example, a timer interrupt unblocks a process that has been sleeping for a certain time. Gandalf enables the quick and timely handling of an interrupt by a guest OS when Gandalf is not running. An interrupt directly invokes the guest Linux's corresponding interrupt handler without Gandalf's intervention. This is possible because Gandalf's spatial resource management scheme maps one physical CPU to one virtual CPU, and all interrupts are guaranteed to go to the same guest Linux.

In contrast, the other VMMs usually intervene interrupts, and notify them to the corresponding guest OS. This approach is required if multiple guest OSes are multiplexed upon a single CPU. An interrupt may be directed to a guest OS that is not currently running; thus, a VMM first handles an interrupt, and decides which guest OS it is delivered to. If the interrupt needs to be delivered to a higher priority guest OS than the current one, the VMM switches the context to the higher priority one, and delivers the interrupt to it. If a VMM intervenes interrupts, it can handle interrupts while it is running because this is analogous to an OS kernel' handling interrupts while the execution is in the kernel. Therefore, there is no problem with making this type of VMMs interruptible.

We did not take the approach that a VMM intervenes interrupts, but chose to deliver interrupts directly to a guest OS. We made this decision because 1) the IA-32 architecture allows such direct interrupt delivery by setting the interrupt descriptor table (IDT) appropriately, and 2) it obviously causes less overhead to deliver interrupts directly rather than a VMM intervenes them. The drawback of this approach is that interrupts cannot be delivered to a guest OS while the execution is in a VMM because interrupts need to be handled in a less privilege ring. This is a limitation brought on by the IA-32 architecture; thus, a special care needs to be taken by a VMM to make it possible. We assumed that the execution time of a VMM is much shorter than that of a guest OS, and opted to leave the VMM uninterruptible. This assumption was, however, incorrect as described below.

2.3 Preliminary Evaluation Results

We show preliminary results obtained by experimenting with a single guest OS to evaluate the overheads incurred by Gandalf. We used the Dell Precision 490 equipped with the Intel Xeon 5130 2.0 GHz processor. We used Linux 2.6.18 with few changes required for the guest OS to run on Gandalf VMM. We performed the same experiments with the native Linux 2.6.18 and paravirtualized XenLinux ⋆2 on Xen 3.1 [1], and compared their results with those of Gandalf.

We employed PMC (Performance Monitoring Counter) to measure the details of the overheads. PMC counts the number of occurrences of the selected events.

⋆1 Only one line of a modification is needed for this change.

⋆2 This version of XenLinux is also based on Linux 2.6.18. Therefore, we used the same version 2.6.18 of the original Linux, XenLinux, and Linux on Gandalf for fair comparisons.

Table 1 PMC event types.

Event Name	Event Type
INST_RETIRED.ANY.P	The number of instructions executed
RESOURCE_STALLS.ANY	The number of stalls
L1I_MISSES	The number of L1 instruction cache misses
L1D_MISSES	The number of L1 data cache misses
ITLB_MISSES	The number of instruction TLB misses
DTLB_MISSES	The number of data TLB misses
PAGE_WALKS.COUNT	The number of page table walks
SEGMENT_REG_LOADS	The number of segment register loads
CYCLES_INT_MASKED	The number of cycles with interrupts disabled

PMC can also be configured to count an event that occurred only when the execution is in Ring 0. By this feature, we can see what portion of the number of events occurred in a VMM, which executes in Ring 0. **Table 1** shows the event names and types that are measured. L1D_MISSES is not provided by PMC, so that L1D_REPL was used instead. The table does not show the number of L2 cache misses. Since the footprints of the benchmark programs fit in the L2 cache, there were no L2 cache misses.

We performed the measurements using three programs, pipe latency, process fork-and-exit and process fork-and-exec, which are not from the LMbench benchmark but function the same as them. We show the results of process fork-and-exit in **Fig. 4** because we can find similarities in the results from all the three programs. The results shown in the figure were normalized relative to the original Linux. The results show that Xen causes more cache and TLB misses than Gandalf. Especially, Xen causes significantly more data TLB misses and page table walks. From the results, we can assume the larger memory footprint of Xen is the major source of its overheads. In other words, the simplicity of Gandalf's design makes its footprints smaller and contributes to its performance.

There is, however, only one event that Xen performs better than Gandalf. The number of the CYCLES_INT_MASKED event is smaller for Xen. It counts the cycles while interrupts are disabled (masked). The larger number of the CYCLES_INT_MASKED event means that hardware interrupt latencies can be longer. Since Gandalf targets embedded systems, such latencies should be shorter; the number of the CYCLES_INT_MASKED event should therefore be improved.

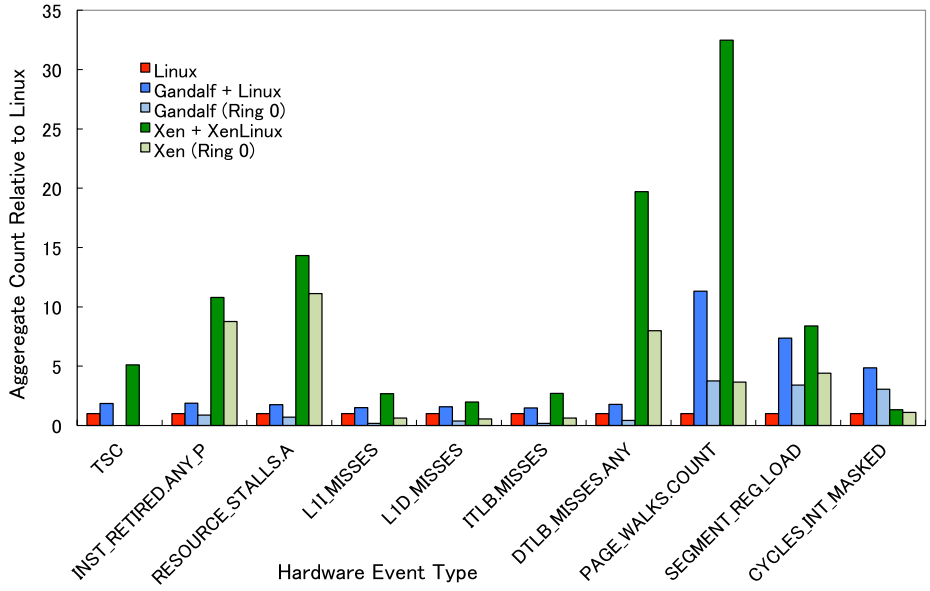


Fig. 4 Preliminary results – fork+exit.

3. Interruptible Gandalf

As described in the previous section, the evaluations using CPU's performance monitoring counters (PMC) revealed Gandalf executes with interrupts disabled for a rather long duration of time. This is because Gandalf handles events that are reported as faults, such as general protection faults and page faults. A guest Linux's execution of a privileged instruction causes a general protection fault, and Gandalf handles the fault to emulate the instruction. When a page fault occurs, Gandalf handles the fault to maintain the shadow page table. If a lower priority process caused a fault and invoked a VMM before the timer interrupt occurred, however, the delivery and handling of the interrupt is delayed because the VMM executes with interrupts disabled. Such a delay of handling an interrupt causes the priority inversion problem. Therefore, it is important for a VMM to be interruptible, so that it can handle interrupts that occur even while the VMM are handling a fault.

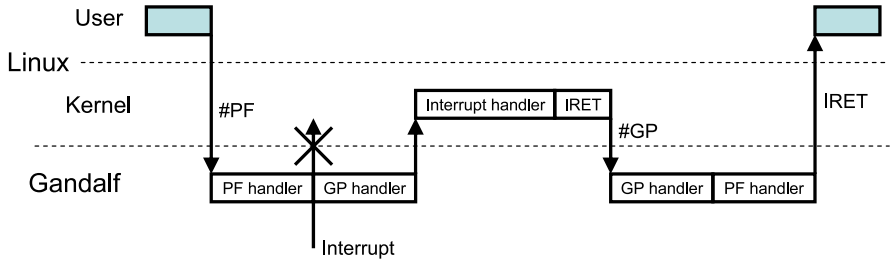


Fig. 5 Example of an execution path to invoke Linux's interrupt handler in order to respond to an interrupt that occurred when Gandalf is running.

We improved the design of Gandalf by making Gandalf interruptible, and solved the design issue that Gandalf executes with interrupts disabled for a rather long duration of time. The rest of this section describes in detail the design and implementation of interruptible Gandalf.

3.1 Rationale

We first investigate the mechanisms to make Gandalf interruptible. Assuming that Gandalf is interruptible, **Fig. 5** depicts a typical execution path by which Gandalf responds to an interrupt that occurred when Gandalf is handling a page fault of a user process. When a user process of Linux causes a page fault, Gandalf's page fault handler is invoked. An interrupt occurs while the page fault handler is still running. Since the corresponding interrupt handler is in the Linux kernel and a CPU does not allow a handler in a lower privilege ring to be invoked, an attempt to invoke the handler in Ring 1, which is for the Linux kernel, causes a general protection fault. Gandalf's general protection fault handler finds that the interrupt caused the fault; thus, it manually invokes the Linux's corresponding interrupt handler. When Linux finishes the interrupt handling, it executes the IRET instruction to return from the handler. Such an execution of IRET again causes a general protection fault because IRET cannot be used to return to the higher privilege ring. Gandalf's general protection fault handler takes this chance to resume the execution of the page fault handler.

This example suggests that, in order to handle interrupts that occurred during Gandalf's execution, Gandalf needs to support the nest of traps because appropriate handling of general protection faults is required during the original trap handling. Specifically, interruptible Gandalf needs to be able to invoke Linux's interrupt handler during Gandalf's execution and to have the handler return to Gandalf to resume its execution. In this scheme, during the execution of Linux's kernel or user process, an interrupt still can directly invoke Linux's interrupt handler without Gandalf's intervention. Since the execution is in Linux for the most of time, it is advantageous to keep the lightweight interrupt handling implemented in Gandalf.

3.2 Invoking Linux's Interrupt Handler

If an interrupt occurs during the execution of Gandalf with interrupts enabled, the invocation of Linux's interrupt handler causes a general protection fault because of the IA-32's protection architecture as described above. Gandalf's general protection fault handler is invoked by two reasons, Linux's execution of privileged instructions and interrupts; thus, it has to be able to differentiate between them and to identify the exact cause of the fault. The handler can distinguish interrupts from the execution of privileged instructions by looking at the error code of a fault. A general protection fault pushed an error code onto the VMM stack, and its value is different for each reason. If it finds the fault was caused by an interrupt, it reads the ISR (In-Service Register) in APIC (Advanced Programmable Interrupt Controller) to obtain the interrupt number; therefore, all information needed to invoke Linux's interrupt handler can be obtained.

Once Gandalf's general protection fault handler obtains the necessary information to invoke Linux's interrupt handler, Gandalf sets up the stacks of Gandalf and the Linux kernel to prepare for the invocation. Both of the stacks need to be manipulated because they carry different information. The preparation takes the following three steps. First, Gandalf saves the current context by copying the current stack frame on the Gandalf stack to the old context save area, which was allocated in advance at boot time (**Fig. 6** (1) save). It also needs to save some additional bytes above the current frame because they are corrupted by the third step, which will be described below. Only one save area is needed because the following interrupts are handled directly in Linux and can avoid general protection faults. Second, Gandalf pushes the interrupted context information onto the Linux kernel stack and creates the structure as if the interrupt directly invoked the Linux's interrupt handler (Fig. 6 (2) copy). The pushed data is used to return

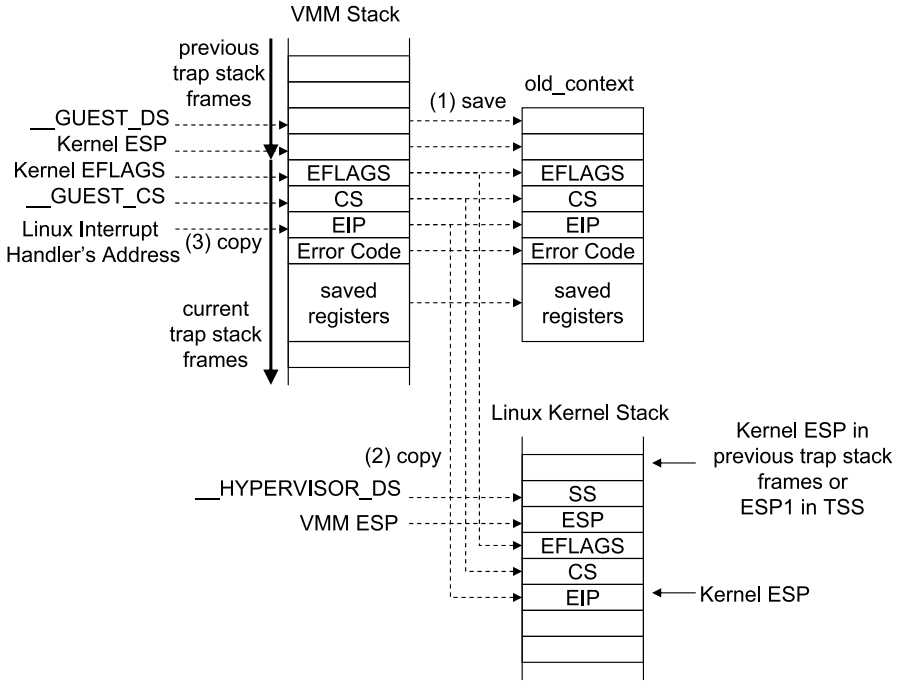


Fig. 6 Manipulation of Gandalf and the Linux kernel stacks for the preparation to invoke Linux's interrupt handler from Gandalf.

to Gandalf after Linux's interrupt handler finished handling the interrupt. The next section describes the returning to Gandalf in detail. Finally, Gandalf sets up the stack frame on the Gandalf stack so that it can upcall Linux's handler using the IRET instruction (Fig. 6 (3) copy). IRET restores the data, such as the instruction pointer (EIP), the stack pointer (ESP), and the text and data segment selectors (CS, DS), which were pushed onto the Gandalf stack by the third step, and then the execution starts from the address specified by EIP at the privilege specified by CS. Since the current implementation pushes the DS and ESP values onto the previous trap stack frame and corrupts 8 bytes, these are also saved in the first step described above.

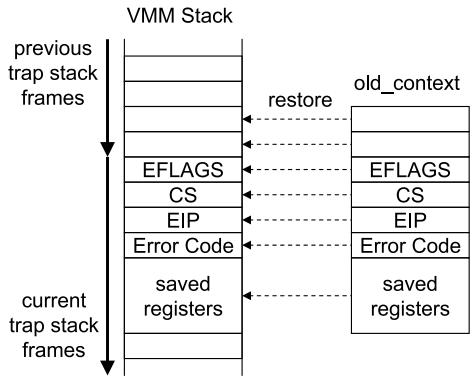


Fig. 7 Stack usage when return to Gandalf.

3.3 Returning to Gandalf

After Linux's interrupt handler finished handling an interrupt, Linux executes the IRET instruction to resume the interrupted execution. When Gandalf invokes Linux's interrupt handler as described above, the execution of IRET with the stack frame created by Gandalf causes a general protection fault. The CS in the stack frame to be used by IRET points to Gandalf's text segment whose privilege is higher than that of the Linux's segment. Since IRET does not allow the execution to return to the higher privilege, Gandalf receives a general protection fault handler and finds that Linux's interrupt handler has finished its execution.

When Gandalf's general protection fault handler finds that the cause of a fault is Linux's execution of IRET and also that there is valid information saved in the old context save area, it determines that is needs to resume the interrupted execution. The information to be restored was saved at the first step to invoke Linux's interrupt handler (Fig. 6 (1) save). Gandalf restores the interrupted context by copying data from the old context save area back to the Gandalf stack (**Fig. 7**) and makes the stack the same as the point when interrupt just occurred. Gandalf then executes IRET to return to the interrupted point and resume the execution.

3.4 Implementation

We implemented the mechanisms to make Gandalf interruptible as described in

the previous sections, and enabled interrupts at the following sections in Gandalf:

- handle_set_pte() hypercall,
- flush_shadow_pgd() hypercall,
- a part of the page fault handler where the shadow page table is updated,
- INVLPG emulator in the general protection handler.

These sections manipulate shadow page tables and the number of executing instructions are large. Making these sections interruptible is therefore considered an effective way to lower the total number of interrupt masked cycles in Gandalf.

The implementation requires complicated stack contents manipulation in order to save and to move around data on the VMM stack. The complication comes from the limitation of the IA-32 architecture, which does not expect an interrupt handler to be executed in a less privilege ring. The generalization is possible by processing interrupts in separate contexts. This is analogous to treating interrupts as threads [11] and also to transforming interrupts to IPC messages [5]. Such a generalization may, however, incur a greater overhead.

4. Performance

We experimented with both the original and modified Gandalf to determine the cost required to make it interruptible and the improvement over the total number of interrupt masked cycles. The experiments were performed under the same environments described in Section 2.3.

4.1 Evaluation with LMbench Microbenchmark

First, we show the results from the LMbench benchmark programs [12] in **Fig. 8** to see the costs required to make Gandalf interruptible. The LMbench consists of a number of benchmark programs that measure the basic operation costs of an OS. We chose three programs, pipe latency, process fork-and-exit, and process fork-and-exec. From the results, we can compare the performance of interruptible Gandalf with the non-interruptible version of Gandalf, the original Linux, and Xen.

The results show that Linux on Gandalf performs slightly slower than the original Linux for those benchmark programs, but much faster than Xen. Although Xen applies paravirtualization to XenLinux for better performance, Gandalf outperforms Xen by its simple and lightweight design and implementation.

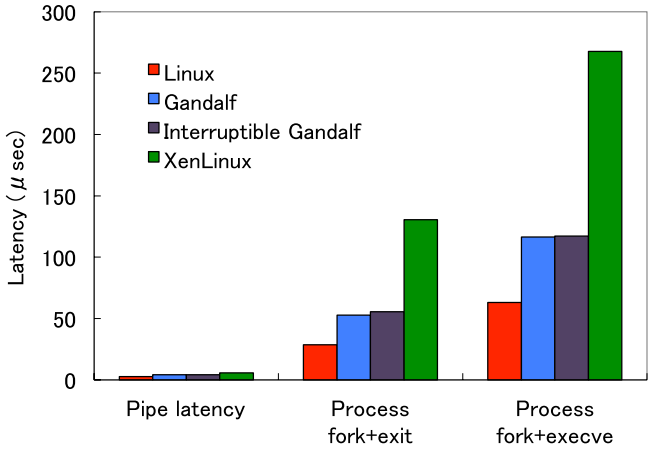


Fig. 8 Basic performance evaluation performed by using the LMbench benchmark programs.

The differences between the interruptible and non-interruptible versions of Gandalf are negligible. Interruptible Gandalf slows down only 5% at most for process fork-and-exit, but it performs almost the same (less than 1%) for the rest of the benchmark programs.

4.2 Interrupt Masked Cycles

We analyzed the differences between the interruptible and non-interruptible versions of Gandalf by using PMC. We focus on the CYCLES_INT_MASKED event, by which PMC counts the cycles when interrupts are disabled (masked). We used the same programs used in Section 2.3 for the measurements. **Figure 9** shows the results that were performed on the both versions of Gandalf and the original Linux. Measurements were performed on both versions of Gandalf to count events that occurred in all protection rings and only in Ring 0. The events in Ring 0 signify that they occurred during the execution of Gandalf only.

The results show that the interrupt masked cycles are significantly reduced on interruptible Gandalf for the process fork-and-exit and process fork-and-exec programs. The results from the pipe latency program are almost the same. As described in Section 3.4, Gandalf currently enables interrupts only during certain sections, which are considered large enough to make the expense of upcalling

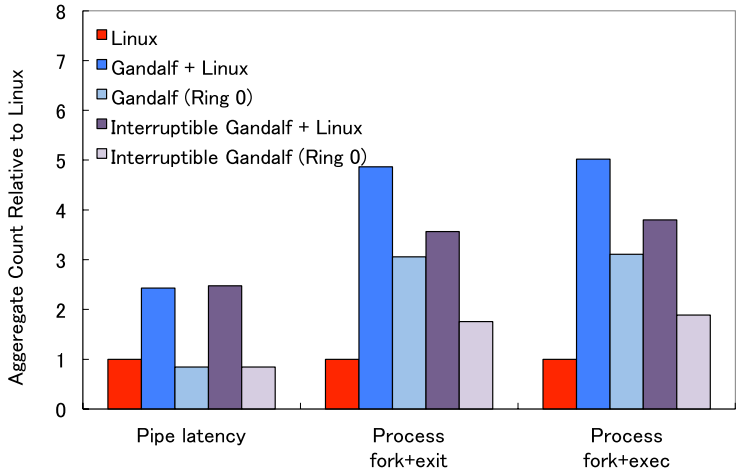


Fig. 9 Comparison of CYCLES_INT_MASKED counts.

Table 2 Results of interrupt latency.

	Native Linux	Gandalf	Interruptible Gandalf	Xen
TSC counts	21,868	22,908	22,977	23,902
latency (μsec)	10.96	11.48	11.52	11.98

Linux's interrupt handler cost-effective. Those sections are mostly related to the manipulation of shadow page tables. The process fork-and-exit and process fork-and-exec programs utilize those sections that are made interruptible. We can therefore see a significant difference. On the other hand, the execution of the pipe latency program does not involve the manipulation of shadow page tables; thus, we do not see any improvement.

4.3 Interrupt Latency

Finally, we show the measurement results of the interrupt latency on the both versions of Gandalf, the original Linux, and Xen, in **Table 2**. We measured the latency from the time the interrupt handler in the Linux kernel starts to handle the interrupt until the time the process waiting for the interrupt resumes its work, using the RTC (Real-Time Clock) device.

Compared with the native Linux, interruptible Gandalf is 5.1% slower while the non-interruptible version of Gandalf is 4.8% slower. The differences between the interruptible and non-interruptible versions of Gandalf are very small and negligible. Although Gandalf is slower than the native Linux, it can respond to interrupts faster than Xen.

The evaluation results described in Section 4 have shown that the effort of making Gandalf interruptible significantly reduced the sections where interrupts are disabled while it does not impact the performance. We, however, need to investigate where we can further reduce the sections where interrupts are still disabled.

5. Related Work

There have been lots of efforts to make OS kernels preemptive in order to improve the real-timeliness of systems. Preemptive kernels can dispatch a higher priority process, which was made runnable by an event, such as an interrupt or a message, while another process is executing in the kernel. Non-preemptive kernels allow another process to be dispatched only at the certain point where the current process finished its execution in the kernel and is returning to the user level. There are mainly two approaches to making kernels preemptive. One is to place multiple preemption points where the current process can be safely preempted. The DEC ULTRIX [2], Sun OS 5.0 [10], and Linux 2.6's CONFIG_PREEMPT option took this approach. The other approach is to handle interrupts in the context of kernel threads and to make such interrupt handling threads schedulable. Sun Solaris [11] took this approach, and there is an effort to incorporate such changes in Linux [14]. This approach, however, requires the significant changes to the kernel software architecture and thus quite a number of modifications in the kernel source code. L4 [5] is a microkernel that converts interrupts into IPC messages, and then threads handle the messages at the user level. L4 microkernel is, however, not itself preemptive. REAL/IX [3] comes between the two approaches.

Our work is somewhat similar to the above efforts to make OS kernels preemptive, supposing an OS kernel is a VMM and a user process is a guest OS. Our work is, however, inherently different since the major components of Gandalf are the handlers of exceptions and faults, of which causes are themselves indivisible. We incorporated the first approach, placing preemption points, to make Gandalf interruptible. Although the approach itself is not new, the architecture of the tar-

get software system is completely different; this work is therefore one more step toward making VMM more suitable for resource constrained embedded systems.

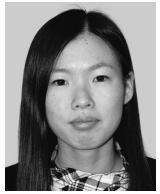
6. Conclusion

We described our approach to enable Gandalf VMM to be interruptible. Although Gandalf was designed to be a lightweight VMM, the detailed performance analysis using PMC showed that Gandalf executes with interrupts disabled for a rather long duration of time. By making Gandalf interruptible, we are able to make VMM based systems more suitable for embedded systems. We analyzed the requirements needed for making Gandalf interruptible, designed and implemented mechanisms to achieve this. The experimental results showed that making Gandalf interruptible significantly reduced a duration of execution time with interrupts disabled while it did not impact the performance.

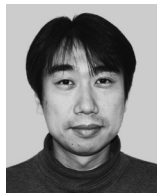
References

1) Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the Art of Virtualization, *Proc. 19th ACM Symposium on Operating System Principles*, pp.164–177 (Oct. 2003).
2) Fisher, T.: Real-Time Scheduling Support in Ultrix-4.2 for Multimedia Communication, *Proc. 3rd International Workshop of Network and Operating System Support for Digital Audio and Video*, LNCS 712, pp.321–327, Springer-Verlag (1993).
3) Furht, B., Parker, J. and Grostick, D.: Performance of REAL/IX-Fully Preemptive Real Time UNIX, *ACM SIGOPS Operating Systems Review*, Vol.23, No.4, pp.45–52 (Oct. 1989).
4) Goldberg, R.P.: Survey of Virtual Machine Research, *IEEE Computer* (June 1974).
5) Hartig, H., Hohmuth, M., Liedtke, J., Schonberg, S. and Wolter, J.: The Performance of μ-Kernel-Based Systems, *Proc. 16th ACM Symposium on Operating System Principles* (Oct. 1997).
6) Intel Corporation: IA-32 Intel Architecture Software Developer's Manual.
7) Ito, M. and Oikawa, S.: Mesovirtualization: Lightweight Virtualization Technique for Embedded Systems, *Proc. 5th IFIP International Workshop on Software Technologies for Future Embedded and Ubiquitous Systems* (*SEUS 2007*), LNCS 4761, pp.496–505, Springer-Verlag (May 2007).
8) Ito, M. and Oikawa, S.: Lightweight Shadow Paging for Efficient Memory Isolation in Gandalf VMM, *Proc. 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing* (*ISORC 2008*), pp.508–515 (May 2008).
9) Ito, M. and Oikawa, S.: Improving Real-Time Performance of a Virtual Machine Monitor Based System, *Proc. 6th IFIP International Workshop on Software Technologies for Future Embedded and Ubiquitous Systems* (*SEUS 2008*), LNCS 5287, pp.114–125, Springer-Verlag (Oct. 2008).
10) Khanna, S., Serbree, M. and Zolnowsky, J.: Realtime Scheduling in SunOS 5.0, *Proc. Winter '92 Usenix Conference*, pp.375–390 (1992).
11) Kleiman, S. and Eykholt, J.: Interrupts as Threads, *ACM SIGOPS Operating Systems Review*, Vol.29, No.2, pp.21–26 (Apr. 1995).
12) McVoy, L. and Staelin, C.: LMbench: Portable Tools for Performance Analysis, *Proc. USENIX Annual Technical Conference*, pp.279–294 (Jan. 1996).
13) Meyer, R. and Seawright, L.: A Virtual Machine Time Sharing System, *IBM Systems Journal*, Vol.9, No.3, pp.199–218 (1970).
14) Real-Time Linux Wiki. http://rt.wiki.kernel.org/
15) Rosenblum, M. and Garfinkel, T.: Virtual Machine Monitors: Current Technology and Future Trends, *IEEE Computer*, pp.39–47 (May 2005).

(Received November 29, 2010)
(Accepted May 14, 2011)
(Released August 10, 2011)



Megumi Ito received her B.E. and M.E. from University of Tsukuba in 2006 and 2008, respectively. She has been a researcher at IBM Research – Tokyo from 2008. Her major research interest is systems software.



Shuichi Oikawa received his B.S., M.S., and Ph.D. degrees in Computer Science from Keio University in 1989, 1991, and 1996, respectively. He has served as an Associate Professor of the Department of Computer Science at University of Tsukuba since 2004. Before joining University of Tsukuba, he worked at Keio University, Carnegie Mellon University, Intel Corporation, Sun Microsystems, and Waseda University. His research interest is systems software including operating systems and virtual machine monitors. He is a member of IPSJ, IEICE, and IEEE.