Improving Real-Time Performance of a Virtual Machine Monitor based System

Megumi Ito[⋆]   Shuichi Oikawa

Department of Computer Science, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki 305-8573, Japan

Abstract. This paper describes our approach to enable Gandalf VMM (Virtual Machine Monitor) to be interruptible. Although Gandalf is shown to be a lightweight VMM, the detailed performance analysis using PMC (Performance Monitoring Counters) showed 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 and ubiquitous systems. We analyzed the requirements to make Gandalf interruptible, designed and implemented the mechanisms to realize it. The experimental results shows that making Gandalf interruptible significantly reduces a duration of execution time with interrupts disabled while it does not impact the performance.

1 Introduction

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

A major barrier of employing VMMs on embedded and ubiquitous 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 paravirtualization, 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 suits resource constrained embedded and ubiquitous systems.

The detailed performance analysis, which was performed by using CPU's performance monitoring counters (PMC), also revealed Gandalf executes with interrupts disabled for a rather long duration of time [8]. This is because Gandalf handles events that

[⋆] She is currently with IBM Research, Tokyo Research Laboratory. Work conducted when she was with University of Tsukuba.

are reported as faults, and such handling of faults is usually done with interrupts disabled. There paper describes our effort to improve Gandalf's real-time performance. We analyzed the requirements to make Gandalf interruptible, designed and implemented the mechanisms to realize it. The experimental results show that making Gandalf interruptible significantly reduces a duration of execution time with interrupts disabled while it does not impact the performance.

The rest of this paper is organized as follows. Section 2 describes the overview of Gandalf VMM. In Section 3 we describe how we made Gandalf interruptible. Section 4 shows the performance of interruptible Gandalf and Section 5 describes the related work. Finally, Section 6 concludes the paper.

2 Overview of Gandalf

This section 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. Fig. 1 shows the structure of a Gandalf VMM based system. Gandalf is a Type-I VMM, which executes 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 execute independently on each virtual machine. 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 if not otherwise specified.

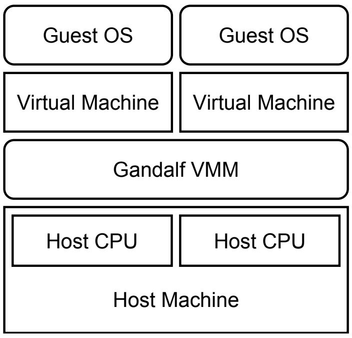

Fig. 1. Structure of Gandalf based system

The IA-32 architecture provides 4 privilege levels (rings) from 0 to 3. A numerically less privilege level is more privileged; thus, Ring 0 is the most privileged. 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 less) 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 uses of privileged instructions cause 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.

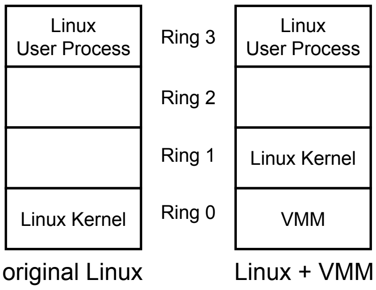

Fig. 2. Privilege level usage

Native Linux kernels normally use all the physical memories 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 improper uses of physical memories. Concerning a virtual address space, there needs to be an area where a VMM resides. Linux kernels, however, normally use all the virtual address space, which overlaps the virtual memory area for the VMM. To avoid

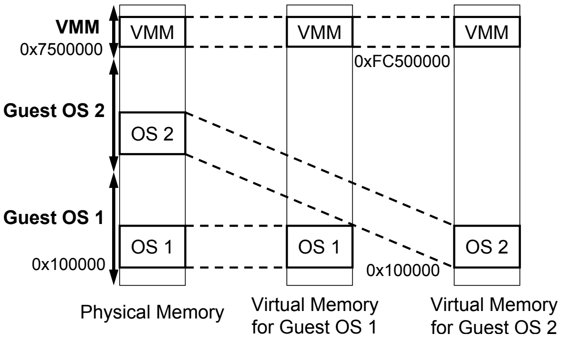

Fig. 3. Memory map

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 for the VMM.

3 Interruptible Gandalf

We have shown that Gandalf is a lightweight VMM from intensive performance evaluations using CPU's performance monitoring counters (PMC) [8]. The evaluations using PMC also 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. It is natural for those faults to be handled with interrupts disabled because the causes of those faults are themselves indivisible.

Embedded systems require quick and timely responses to interrupts. An interrupt can be an event that processes have been waiting for; thus, in that case, it 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 it is not running. An interrupt invokes the guest Linux's corresponding interrupt handler directly without Gandalf's intervention. This is possible because Gandalf's spatial resource management scheme maps one physical CPU to one virtual CPU and it is guaranteed that all interrupts go to the same guest Linux.

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 an interrupt that occurs even while the VMM are handling a fault.

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

We successfully made Gandalf interruptible and reduced a duration where Gandalf executes with interrupts disabled. 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. Supposing Gandalf is interruptible, Fig. 4 depicts an example execution path that Gandalf responds to an interrupt, which occurred when Gandalf is handling a page fault of a user process. When Linux's user process 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.

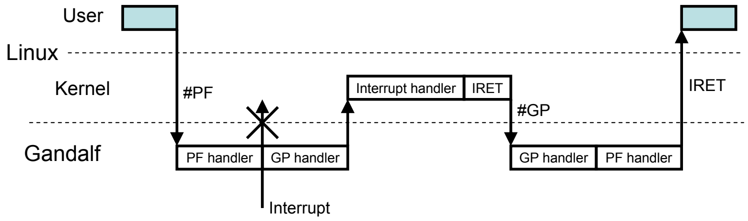

Fig. 4. Example of an execution path to invoke Linux's interrupt handler in order to respond to an interrupt that occurred when Gandalf is running

This example suggests that, in order to handle interrupts occurred during Gandalf's execution, Gandalf needs to support the nest of traps because the 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 the 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 the boot time (Fig. 5 (1) save). It also need 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. 5 (2) copy). The pushed data is used to return 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. 5 (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, they are also saved in the first step described above.

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, of which privilege is higher than 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 finished its execution.

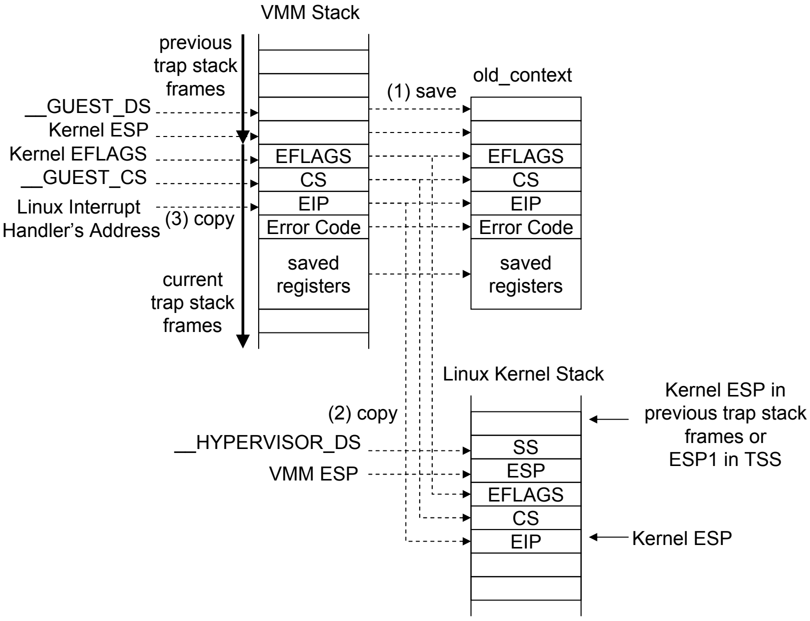

Fig. 5. Manipulation of Gandalf and the Linux kernel stacks for the preparation to invoke Linux's interrupt handler from Gandalf

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. 5 (1) save). Gandalf restores the interrupted context by copying data from the old context save area back to the Gandalf stack (Fig. 6) 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 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; thus, making these sections interruptible are considered to be effective in order to decrease the total number of interrupt masked cycles in Gandalf.

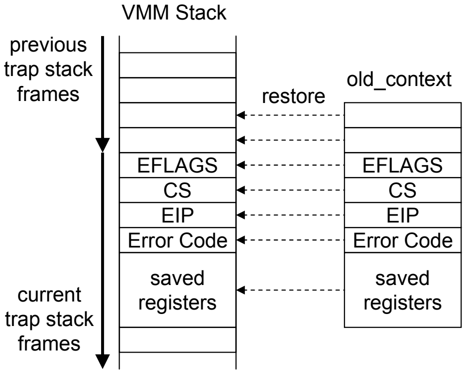

Fig. 6. Stack usage when return to Gandalf

4 Performance

We experimented with a single guest OS to evaluate the costs to make Gandalf interruptible and its improvement in the total number of interrupt masked cycles in 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 employed PMC (Performance Monitoring Counter) to measure the total number of interrupt masked cycles. The experiment to measure the interrupt latency was also performed. 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.

4.1 Evaluation with LMbench Microbenchmark

First, we show the results from the LMbench benchmark programs [11] in Fig. 7. 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, to make the baseline of the performance. 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.

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.

[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.

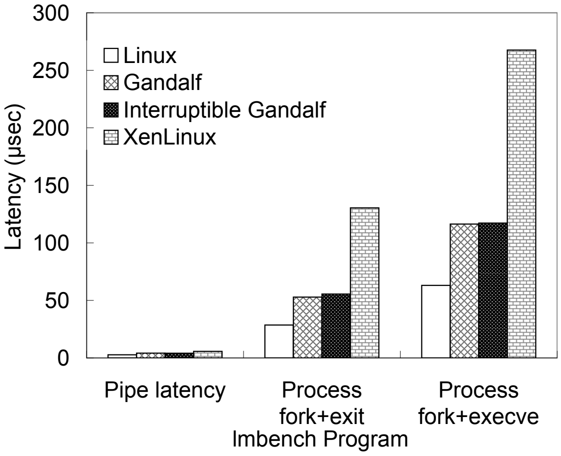

Fig. 7. Basic performance evaluation performed by using the LMbench benchmark programs

4.2 Interrupt Masked Cycles

We further analyze the differences between the interruptible and non-interruptible versions of Gandalf by using PMC (Performance Monitoring Counter) provided by the target CPU. PMC counts the number of occurrences of the selected events. 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. There are many kinds of events PMC can count, such as L1 and L2 cache misses, TLB misses, and so on. We used PMC to investigate why Gandalf outperforms Xen and found that the memory footprint is the major source of overheads in Xen [8].

This paper focuses on the CYCLES_INT_MASKED event, by which PMC counts the cycles when interrupts are disabled (masked). We used the three programs that are not from the LMbench benchmark but function the same as them. Fig. 8 shows the results of the measurements that were performed on the both versions of Gandalf and the original Linux. For the both versions of Gandalf, the measurements were performed to count events occurred in all protection rings and only in Ring 0. The events in Ring 0 means 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, so that the costs of upcalling Linux's interrupt handler pays. Those sections are mostly related to the manipulation of shadow page tables. The process fork-and-exit and process fork-and-exec programs exercises those sections that are made interruptible. Therefore, we see the significant difference. On the other hand, the

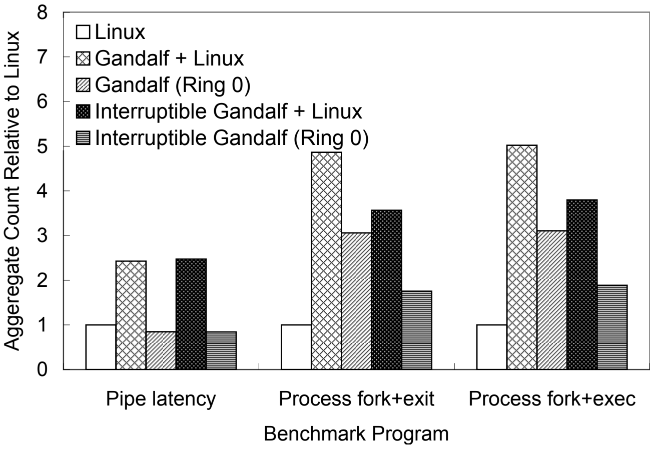

Fig. 8. Comparison of CYCLES_INT_MASKED counts

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 Fig. 1. 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.

Table 1. Results of interrupt latency

	Native Linux	Gandalf	Interruptible Gandalf	Xen
TSC counts	21868	22908	22977	23902
latency (μsec)	10.96	11.48	11.52	11.98

Comparing 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 the interrupt 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 make 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 [9], 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 [10] took this approach, and there is an effort to incorporate such changes in Linux [13]. 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 convert interrupts into IPC messages, and then threads handle the messages at the user level. L4 microkernel itself is, however, not 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 a VMM are the handlers of exceptions and faults, of which causes are themselves indivisible. We incorporated the first approach, placing preemption points, to make our VMM interruptible. Although the approach itself is not new, the architecture of the target software system is completely different; thus, this work is a step to make VMMs more suitable for embedded systems.

6 Conclusion

We described our approach to enable Gandalf VMM to be interruptible. Although Gandalf was shown 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 and ubiquitous systems. We analyzed the requirements to make Gandalf interruptible, designed and implemented the mechanisms to realize it. 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. We will further investigate where we can reduce the sections where interrupts are currently disabled.

References

1. 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.

2. T. Fisher. Real-Time Scheduling Support in Ultrix-4.2 for Multimedia Communication. In *Proceedings of the 3rd International Workshop of Network and Operating System Support for Digital Audio and Video*, Springer-Verlag LNCS 712, pp. 321-327, 1993.
3. B. Furht, J. Parker, and D. Grostick. Performance of REAL/IX-Fully Preemptive Real Time UNIX. *ACM SIGOPS Operating Systems Review*, 23 (4), pp. 45-52, October 1989.
4. R. P. Goldberg. Survey of Virtual Machine Research. IEEE Computer, June 1974.
5. H. Hartig, M Hohmuth, J. Liedtke, S. Schonberg, and J. Wolter. The Performance of μ-Kernel-Based Systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, October 1997.
6. Intel Corporation. IA-32 Intel Architecture Software Developer's Manual.
7. M. Ito and S. Oikawa. Mesovirtualization: Lightweight Virtualization Technique for Embedded Systems. In *Proceedings of the 5th IFIP International Workshop on Software technologies for future Embedded and Ubiquitous Systems (SEUS 2007)*, Springer-Verlag LNCS 4761, pp. 496-505, 2007.
8. M. Ito and S. Oikawa. Lightweight Shadow Paging for Efficient Memory Isolation in Gandalf VMM. In *Proceedings of the 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2008)*, May 2008. (to be appeared)
9. S. Khanna, M. Serbree, and J. Zolnowsky. Realtime Scheduling in SunOS 5.0. In *Proceedings of the Winter '92 Usenix Conference*, pp. 375-390, 1992.
10. S. Kleiman and J. Eykholt. Interrupts as Threads. *ACM SIGOPS Operating Systems Review*, 29 (2), pp. 21-26, April 1995.
11. L. McVoy and C. Staelin. LMbench: Portable Tools for Performance Analysis. In *Proceedings of the USENIX Annual Technical Conference*, pp. 279–294, January 1996.
12. R. Meyer and L. Seawright. A Virtual Machine Time Sharing System. IBM Systems Journal, 9 (3), pp. 199–218, 1970.
13. Real-Time Linux Wiki. `http://rt.wiki.kernel.org/`.
14. M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. IEEE Computer, pp. 39–47, May 2005.