Mesovirtualization: Lightweight Virtualization Technique for Embedded Systems

Megumi Ito Shuichi Oikawa

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

Abstract. These days, embedded and ubiquitous devices are becoming feature rich, and multiprocessor architectures for those devices are on the horizon. In order to utilize the resources of multiprocessor systems efficiently and securely, virtual machine monitors (VMMs) have been common among servers and desktop systems. The same can be applied if the cost of virtualization becomes much less expensive. In this paper, we introduce *mesovirtualization*, a new lightweight virtualization technique. Mesovirtualization makes VMMs smaller and requires only a few modifications for the guest operating system (OS) source code. We designed and implemented a VMM named Gandalf according to mesovirtualization. Our experimental results show that Linux on Gandalf performs better than XenLinux. Therefore, mesovirtualization makes virtualization environments suitable for embedded and ubiquitous devices.

1 Introduction

Expectations for virtualized execution environments to be used in embedded and ubiquitous devices are becoming higher and higher day by day. While the provision of secure and reliable, yet efficient execution environments is a must for those devices, users' desire for using applications of their own choices is rapidly growing. In order to deal with both requirements, safe programming languages, Java for most cases, have been employed. Such a language based solution restricts applicable applications because of its performance limitation.

Servers and desktop systems adopt virtual machine monitors (VMMs) [6] for mostly the same requirements. Figure 1 shows the structure of a VMM with two guest operating systems (OSes). The physical machine underlying the VMM is a host machine. The VMM operates directly on top of the host machine. Guest OSes can use functions of the host machine only via virtual machines (VMs) realized by the VMM. Because the VMM constructs a VM for each guest OS by virtualizing the functions of a host machine, the guest OSes can operate independently for better security and reliability. On the other hand, because there is no intervention needed to execute applications code on a guest OS, the execution performance of applications on the VMM is much better than that of safe programming languages, such as Java.

A major barrier of applying such virtualization to embedded and ubiquitous devices is the limited resources of those devices. Therefore, we propose a new lightweight virtualization technique, *mesovirtualization*, in order to enable virtualization on them. Mesovirtualization does not require a huge VMM as full virtualization and a huge amount

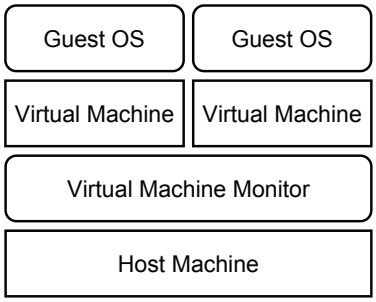

Fig. 1. The Structure of a Virtual Machine Monitor based System

of modifications to the guest OS source code as paravirtualization. Mesovirtualization provides sufficiently virtualized environments for guest OSes with fewer overheads.

This paper describes the design and implementation of a VMM, named Gandalf, which was constructed according to mesovirtualization. It currently operates on x86 processors, and two Linux 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 shows that the cost to bring up a guest OS on Gandalf is significantly fewer than paravirtualization. Our experimental results show that Linux on Gandalf performs better than XenLinux; thus, Gandalf is an efficient and lightweight VMM that suits resource constraint embedded and ubiquitous devices.

We have done two other studies on Gandalf. First, we applied Gandalf to construct a Linux/RTOS hybrid environment, which enables two OSes, Linux and RTOS, to coexist on Gandalf [10]. Second, we also have an experimental implementation that uses only two protection levels of x86 processors since processor architectures with two protection levels are more common among embedded processors. Those studies also support the high feasibility of Gandalf to be used on embedded and ubiquitous devices. This paper focuses on the rationale of mesovirtualization.

1.1 Related Work

There are two well-known techniques to virtualize a physical environment to support several OSes on the same machine. One technique is full virtualization, and the other is paravirtualization. Each of them has advantages and disadvantages. Full virtualization provides VMs that are identical to a host machine for guest OSes. In this case, guest OSes do not require any modifications because VMMs create VMs which works in the same way as the host machines from guest OSes' point of view. However, in order to virtualize the whole ability of the host machine, VMMs become huge and complicated; thus, the cost to virtualize a physical environment is expensive. IBM VM [2] is one of VMMs using full virtualization. IBM VM implements many of virtualization functions in its proprietary hardware in order to lower the cost of full virtualization.

On the other hand, paravirtualization does not virtualize the whole ability of a host machine, but rather artificially creates VMs which are advantageous to guest OSes for

efficiency. VMMs become smaller and simpler; thus, VMs can achieve higher performance. However, it requires a huge amount of modifications to the guest OS source code because the VMs on which guest OSes run are different from the host machine. Xen [1] is one of VMMs using paravirtualization. While the performance of Linux on Xen, called XenLinux, is comparable with the original Linux on a physical machine for some workloads, it requires a lot of modifications to the guest OS source code.

Mesovirtualization differs from both full virtualization and paravirtualization in many ways. Mesovirtualization enables small and simple VMMs to support guest OSes, which makes it possible to provide virtual machines to guest OSes with higher performance. The number of modifications to guest OS source code which mesovirtualization requires is significantly less than paravirtualization. It means that we can use an OS as a guest OS with much few costs.

Pre-virtualization [8] is a new virtualization technique that addresses the cost to bring up an OS on a VMM. It needs much less modifications to a guest OS than paravirtualization, and eases the adoption to a different VMM by having a virtualization module that transforms the standard platform API into the VMM API. Although pre-virtualization shares one of the goals of mesovirtualization, it does not address the code size and memory footprint of a VMM as mesovirtualization does.

The use of virtual machine monitors is not only the way to execute an OS above the processor's most privileged level. Microkernels, such as Mach3 [4] and L4 [5], provides simplified abstractions to run OSes as their applications. More recently, even Linux showed the capability to execute another Linux as its application [3]. The approach of running OSes as applications is close to paravirtualization in terms of the high costs of modifications needed for the guest OS source code.

Partitioning an OS environment into multiple management domains with independent name spaces is also possible. FreeBSD jails [7], Linux VServer, and Solaris containers are the examples. The key difference between those OS partitioning and virtualization is that the OS partitioning has a single kernel shared among multiple domains while virtualization runs multiple kernels, which are not shared. Such sharing of a single kernel makes difficult to maintain predictability required by embedded systems.

1.2 Paper Organization

The rest of this paper is organized as follows. In Section 2 we propose mesovirtualization, a new virtualization technique we introduced to build a lightweight VMM. Section 3 describes the design and implementation of a VMM, named Gandalf, which we built according to mesovirtualization. Section 4 describes the current status of its development, and also shows its evaluation results. Finally, Section 5 concludes the paper.

2 Mesovirtualization

We propose mesovirtualization, a new lightweight virtualization technique to construct a VMM. Mesovirtualization opts to modify a few parts of the guest OS source code in order to enable lightweight configuration of a VMM, but the cost of modifications can be kept as low as possible to make the modifications easily manageable. Therefore,

it does not require the complicated work typically needed for full virtualization and paravirtualization. We must configure huge VMMs for full virtualization and modify a huge amount of the guest OS source code for paravirtualization. Mesovirtualization does not require such complicated work, yet it can provide guest OSes with sufficient virtualization environments, in which guest OSes can manage their environments, use processors, memory and devices as if they run on a physical machine.

Mesovirtualization is based on the principle of minimalism. We do not need to virtualize the entire of the host machine to provide identical environments to guest OSes as full virtualization. We do not need to modify many of the guest OS source code to trap into a VMM and to handle it in the VMM as paravirtualization. Mesovirtualization is a technique which supports guest OSes just enough to run it on a VMM. For some parts of the host machine that are considered safe to be dedicated or shared, a VMM does not virtualize these parts and allows guest OSes touch them directly. This rationale keeps a VMM as simple as possible, so that the code size and memory footprint of the VMM small and also the costs of virtualization can be kept cheap; thus, it can be used on embedded and ubiquitous systems, of which computing resources are not as rich as desktop and server systems.

One significant characteristic of mesovirtualization is how a VMM handles sensitive instructions used in guest OSes. While they are emulated by a VMM very much like in full virtualization, only the essentials are emulated. There are some cases that sensitive instructions which are not emulated by a VMM produces unexpected results for a guest OS. Such cases are actually very rare. Rather than having every sensitive instruction changed to trap to a VMM and handled it with hard work, mesovirtualization modifies guest OS source code a little and manages without causing an interrupt to a VMM.

Such a characteristic leads to a lightweight VMM. Because it does not need to virtualize the full ability of the host machine, such a VMM is released from the jobs to spend a number of lines providing exactly the same machine to guest OSes. It also leads to the reduction of VMM's use of processor time, which makes it possible to provide higher performance to guest OSes.

A decrease in the number of modifications of guest OS source code is another characteristic. While it requires just a few modifications concerned to memory management, it does not need to change the most parts, which is required in Xen. Because we modify a few lines of the source code concerned to sensitive instructions rather than change all of them, we can decrease the modifications to the guest OS source code. Such characteristic also reduces the cost to use an OS as a guest OS on a VMM. Although a guest OS must be modified a little before bringing it up on a VMM, the cost of the required modifications is much less than paravirtualization.

3 Design and Implementation

According to mesovirtualization, we designed and implemented a VMM named Gandalf. Gandalf currently operates on x86 processors, and provides virtual machines for Linux. The next section describes the architectural design of Gandalf, and the following section describes its implementation. The last section shows the modifications we made to the Linux source code.

3.1 Architectural Design

In order to execute guest OSes on a VMM and to make guest OSes not to invade it, it is essential to control the behavior of guest OSes. It can be achieved by combining the following two means. One is to use the ring protection architecture with 4 privilege levels of x86 processors. A VMM can control guest OSes if it has a higher privilege level than guest OSes. The other is the segmentation and paging architecture for memory management. A VMM can manage accesses of guest OSes to memory by setting limitation to the available memory for guest OSes. The VMM decides the physical memory partitions and constructs the first mapping. It also manages the page directory pointer in CR3 register, which implies the base address of the page directory.

An x86 processor employs the ring protection architecture with 4 different privilege levels from Ring 0 to 3. Ring 0 is the most privileged and Ring 3 is the least privileged; thus, guest OSes usually use Ring 0 for kernel and Ring 3 for user processes. In our design, Gandalf uses Ring 0 because it has the strongest privilege in the system so that Gandalf can manage the behavior of guest OSes. Therefore, we changed guest OSes kernel to operate in Ring 1 from Ring 0.

We also modified the segment limit and the privilege level in the segment descriptors of guest OSes, so that the guest OSes do not access the memory for Gandalf and do not use Ring 0 used by Gandalf. Both the segment limit and the privilege level are essential to manage the access to the memory.

Gandalf is in the role of setting up all the memory for itself and guest OSes. It statically partitions the physical memory for itself and guest OSes. It allocates the top most part of the physical memory for itself and the other parts for guest OSes. It changes the start and end addresses of the usable memory for guest OSes to those of the allocated physical memory. Gandalf sets up the first version of the page table using the allocated memory for each OS, and enables paging before booting it. Gandalf maps itself on the top of the virtual memory in every Guest OS's virtual memory space. This first mapping emulates the physical memory; thus, it enables every guest OS starts from the same address as the physical memory. Such provision of the initial mapping reduces the guest OS modifications. In general OSes, they boot with paging disabled and then enable paging during the boot sequence. A guest OS on Gandalf boots with paging enabled in order to make the address of the guest OS look the same as it boots directly on the processor. After guest OSes starts running, they are responsible for the most part of the memory management. Except for setting a new page directory pointer to CR3 register and managing a page fault caused in Gandalf, guest OSes care for the page table.

As far as the memory management is concerned, Gandalf is invoked only to handle general protection faults and page faults after guest OSes starts running. Guest OSes' attempts to execute privileged instructions cause general protection faults, and Gandalf emulates them. There are two cases for page faults. Page faults caused by a guest OS can be handled only by the guest OS; thus, Gandalf simply passes the control back to it. Gandalf handles page faults caused by itself. When guest OSes attempt to set a new page directory pointer to CR3 register of the processor in order to change a page table, a general protection fault is reported because the instructions to write to control registers

are privileged instructions. In response to the general protection fault, Gandalf takes control and sets the page directory pointer appropriately.

3.2 Gandalf

Based on the mesovirtualization technique and the architectural design described above, we implemented a VMM named Gandalf. By employing mesovirtualization, Gandalf provides noticeably lightweight virtual environments to guest OSes. As we described in Section 2, Gandalf does not virtualize the entire ability of a host machine. Such a decision decreases the number of the interactions between guest OSes and Gandalf. It enables guest OSes to process most of their jobs without Gandalf's interventions; thus, it leads to the reduction of the virtualization overheads. Since Gandalf targets on a multiprocessor systems, their support is included.

Gandalf first sets up the environment for one guest OS, and builds the environments for the other guest OSes later. It is done during the initialization phase before the first guest OS starts booting. The initialization module has two sub-modules, the setup sub-module and the SMP sub-module. The setup sub-module is executed for every guest OS. On the other hand, the SMP sub-module is executed only once. In the setup sub-module, Gandalf creates an E820 memory map for a guest OS based on the multiboot information, which is a collection of structures containing physical memory map information provided by a boot loader. It changes the start and end addresses of the E820 memory map to the allocated physical memory to the guest OS. The arguments to the guest OS kernel passed from the boot loader are also copied for it. Because the virtual address of the E820 memory map and the arguments used by a guest OS are fixed to static addresses, Gandalf sets up the memory map and the arguments to be placed at the same virtual addresses for every guest OS. Gandalf also relocates each guest OS and its modules to individual memory regions.

In SMP sub-module, Gandalf wakes up the other processors in turn and starts each processor to boot assigned guest OS. In order to wake up the other processors, Gandalf checks the SMP configuration table and sends startup inter-processor interrupts to them.

3.3 Guest OS Modifications

In order to execute guest OSes on Gandalf, we need to modify only a small number of lines of their source code. The cost of such modifications is much less than building a full virtualization VMM or guest OSes' modifications for paravirtualization. The modifications are concerned to three points, the segment descriptor, the judgment of the privilege level, and the memory management. The segment descriptors include the segment limit and the privilege level, which are especially important for Gandalf to manage memory access. The problem on judging the privilege level occurs due to the changes we made in the segment descriptors. There are also some issues in guest OSes memory management because it usually assumes the physical memory is available from the address `0x0`.

The first modification is made to the segment descriptors. As we mentioned in Section 3.1, we need to change the value of the segment limit and the privilege level to avoid guest OSes invading Gandalf. We modified the segment limit from `0xffffffff` to

0xfc400000 and the privilege level from 0 to 1 so that guest OSes do not access the Gandalf's memory accidentally or intentionally and interfere with its processing. Either the segment limit or the privilege level in the segment descriptors affects to the management of the memory access.

Secondly, we changed the judging value used to examine the privilege level of the trapped execution in order to decide if it executed in kernel mode or user mode. In order to judge it, guest OSes perform a logical AND operation on the saved privilege level and 3 as shown in the following pseudo code:

```
if (regs->xcs & 3) { /* for user mode */ }
```

In this example, the saved privilege level of the previously executed code segment is stored in xcs. The result of the logical AND will be 0 in case the privilege level is 0. In this case the guest OS executed in kernel mode; thus, the code in the braces is not executed. On Gandalf, however, the privilege level of the kernel mode in guest OSes is changed to 1. It changes the result of the logical AND, and leads to taking a wrong action. The result remains to be 0 if the judging value was 2. In order to decide the trapped execution mode correctly, we changed such judging values in guest OSes.

Finally, we modified several parts of the guest OS source code concerned to the physical memory management. We changed the codes in setting up the page table, initializing memory zone sizes, and converting the physical address to/from virtual address. Guest OSes usually assume that the physical memory is available from the address 0x0, which causes a problem if the memory starts from another address. On Gandalf, every OS except for one are allocated a physical memory region that starts with a different address. In order to deal with this problem, we added a hypercall to guest OSes for the purpose to ask Gandalf the actual start address of the physical memory it allocated. Guest OSes use this address to construct the page table, initialize zone sizes, and convert physical/virtual address.

4 Current Status and Evaluation

In this section, we describe the current status of Gandalf and show its evaluation results. We evaluated the basic cost of modifying a guest OS to boot on Gandalf. We also measured the costs of issuing a null hypercall and processing a privileged instruction, compared with Xen. Finally, we describe the evaluation result using a benchmark.

4.1 Current Status

We implemented Gandalf from scratch on x86 processors and brought up the Linux OS as a guest OS on it. We first used a single processor system for the development, and moved to a dual processor system after the Linux OS on Gandalf started working on a single processor system. Currently, we can have two configurations. One is that a single Linux OS as a guest OS on a single processor system, and the other is that two Linux OSes as guest OSes on a dual processor system.

Since a dual processor system configuration has not been matured enough, the quantitative evaluation was performed on a single processor system configuration.

Table 1. The Costs of Modifications

	Modified parts	Modified lines
Single Linux OS	15 parts	28 lines
Two Linux OSes	28 parts	73 lines

4.2 Qualitative Evaluation

This section presents the cost of modifying the Linux kernel in order to bring it up on Gandalf. The version of the Linux kernel we used is 2.6.12.3. Table 1 shows the number of parts and lines we modified or added to the Linux source code in order to use it as a guest OS. We evaluated the modification cost for two cases. One is when only one Linux OS runs on Gandalf and the other is when two Linux OSes run simultaneously. The former contains the modifications concerned to segment descriptors and the judging value to examine the privilege level. The latter includes the modifications concerned to memory management in addition to the former modifications.

The results show that the cost to modify the Linux source code is obviously very few. Only 28 lines for 15 parts of modifications are enough to run one Linux OS on Gandalf, and it requires no more than 73 lines for 28 parts to modify to execute two Linux OSes simultaneously. In contrast, Xen requires 2995 lines of modifications to use a Linux OS as its guest OS [1], of which cost is more expensive than Gandalf by two orders of magnitude.

Table 1 also shows that the required modifications for the single Linux OS case are fewer than those for the case of two Linux OSes, which includes the changes to deal with different physical memory start address. There can be other OSes, of which kernel architecture allows physical memory starting with various addresses. If we use such an OS as a guest OS, the modifications we added to the parts for memory management in the Linux source code is not required, therefore the cost of modifications will be fewer than the results in Table 1.

Please note that those modifications need to bring up Linux on Gandalf were made at very obvious places in the Linux source code. Although we made those modifications by hand, it should not be too difficult to make necessary modifications semiautomatically.

4.3 Quantitative Evaluation

In this section, we present the quantitative evaluation of Linux on Gandalf. All measurements reported below were performed on the Dell Precision 470 Workstation with Intel Xeon 2.8GHz CPU.[1] Hyper-threading was turned off, so that all measurements were performed on a single CPU.

We first measured the basic performance related to running a guest OS on a VMM. We measured the costs of issuing a hypercall, and processing a privileged instruction. Table 2 shows the measurement results obtained from Xen and Gandalf. We used cycle counts obtained from RDTSC instruction for these measurements on both Xen and

[1] 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.

Table 2. Basic Performance Comparisons

	Xen	Gandalf
Null Hypercall	0.43 μ sec	0.37 μ sec
Ignored Privileged Instruction	N/A	0.56 μ sec

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.

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 a difference. It is more interesting 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, 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 will take longer to decode and fetch an emulating instruction.

Finally, in order to evaluate our mesovirtualization method used for Linux, we ran several programs included in lmbench benchmark suite [9]. Figure 2 (a) and (b) 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.

The measurement results show that our mesovirtualization method reduces the runtime costs significantly as a Linux OS on Gandalf outperforms XenLinux in all cases.

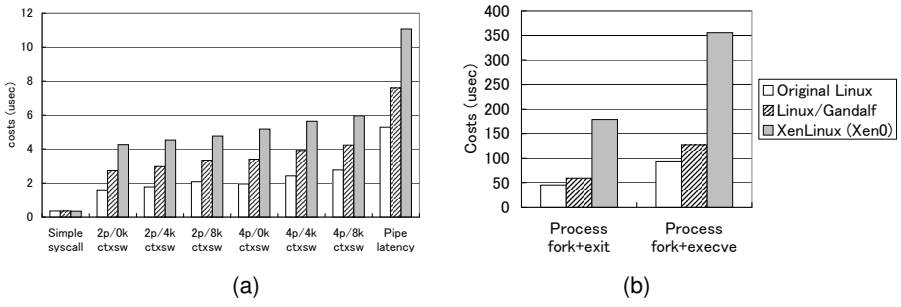

Fig. 2. Linux Performance Comparison

The costs of process fork and exec are even close to the original non-virtualized Linux and significantly better than XenLinux.

5 Conclusion

We introduced mesovirtualization, a new technique that enables lightweight virtualization. It does not require a huge VMM as full virtualization and a huge amount of modifications to the guest OS source code as paravirtualization. Mesovirtualization provides sufficient virtualized environments for guest OSes without complicated work; thus, it makes a whole system more reliable. According to mesovirtualization, we implemented a lightweight VMM, Gandalf. It currently operates on x86 processors and two Linux OSes successfully run on it as guest OSes. The number of the modified parts and lines shows that the cost to modify the Linux source code to bring up Linux OSes on Gandalf is significantly few. The performance evaluations show that the cost for virtualization is also reduced. From the evaluation results, we conclude Gandalf makes virtualization environments suitable for embedded and ubiquitous devices.

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. R. J. Creasy. The Origin of the VM/370 Time-Sharing System. IBM Journal of Research and Development, 25 (5), 1981.
3. J. Dike. A User-mode Port of the Linux Kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*, October 2000.
4. D. Golub, R. Dean, A. Forin, and R. Rashid. UNIX as an Application Program. In *Proceedings of the USENIX Summer Conference*, June 1990.
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. R. P. Goldberg. Survey of Virtual Machine Research. IEEE Computer, June 1974.
7. P. Kamp and R. Watson. Jails: Confining the Omnipotent Root. In *Proceedings of the 2nd International System Administration and Networking Conference*, May 2000.
8. J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie, and G. Heiser. Pre-Virtualization: Slashing the Cost of Virtualization. Fakultät für Informatik, Universität Karlsruhe, Technical Report 2005-30, November 2005.
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. S. Oikawa, M. Ito, and T. Nakajima. Linux/RTOS Hybrid Operating Environment on Gandalf VMM. In *Proceedings of the 2006 IFIP International Conference on Embedded and Ubiquitous Computing*, Springer-Verlag LNCS 4096, pp. 287–296, 2006.