A Study on Asymmetric Operating Systems on Symmetric Multiprocessors

Yu Murata, Wataru Kanda, Kensuke Hanaoka,
Hiroo Ishikawa, and Tatsuo Nakajima

Department of Computer Science, Waseda University
{murata,kanda,kensk,ishikawa,tatsuo}@dcl.info.waseda.ac.jp

Abstract. This paper proposes a technique to achieve asymmetric multiple OSes environment for symmetric multiprocessors. The system has a host OS and guest OSes: L4 microkernel and their servers run as the host OS, and modified Linux runs as the guest OS. OS manager which is one of the servers on the host OS manages the guest OSes. Our approach avoids a lot of execution overheads and modification costs of the guest OSes because the each OS can control hardware directly without any virtualization. The results of the evaluation show that our system is much better than the existing virtual machine systems in point of the performance. In addition, a guest OS in our system requires a few amount of modification costs. Consequently, the experiments prove that our system is a practical approach for both performance and engineering cost sensitive systems.
Key words: Operating Systems, Symmetric Multiprocessors, Multiple OSes Environment, InterProcessor Interrupts, InterOS Communications

1 Introduction

Virtualization technology has become popular again because of the hardware improvement. The remarkable feature of virtual machine technologies is multiple operating systems (OSes) can run on a single machine. A virtualization technology principally consists of a virtual machine monitor (VMM) and some virtual machines (VMs). A VMM provides VMs which virtualize hardware for OSes run on them. If a VMM provides multiple VMs, multiple OSes can run on them concurrently. In general, an OS which runs on a VM is called a guest OS.

On the other hand, virtual machine approaches require higher hardware performance. Some techniques have been proposed to reduce overheads by modifying a guest OS for a specific VMM such as Xen, but they need expensive cost for modification. Therefore, it is significant to consider the trade-off between performance and engineering cost.

Various approaches have been proposed to achieve multiple OSes environment on a single machine. One of the advantages of running multiple OSes is efficient use of hardware resources. It is usually more efficient to run two OSes on a single machine than to run them on two machines. Because recent hardware has a amount of resources such as memory and hard disk, an OS can't always use

all resources in a machine. Running more than two OSes on a single machine, such unused resources can be assigned to another OS. As a result, you can save the number of hardware. In addition, running multiple OSes concurrently improves the reliability of systems. If there are multiple OSes running on a system, you can access their services even when one of the OSes doesn't work. Thus, multiple OSes environment makes the system reliable.

We developed a system which achieves multiple OSes environment on a single machine. The system is named SIGMA system. SIGMA system doesn't require any specific hardware but only requires a symmetric multiprocessor (SMP) which is generally used in a personal computer. SIGMA system assigns one OS to one of the processors. We can say this environment is asymmetric because different kinds of OSes run on each processor in SIGMA system. One of the most important characteristics is that an OS of SIGMA system achieves not only the minimum engineering cost but also the little performance degradation although performance and engineering cost are trade-off relationships in the virtual machine technology.

We ran the benchmark software on its OS to evaluate the performance of SIGMA system and proved its advantage in performance. The evaluation result is actually significant because evaluation of asymmetric environment has hardly been performed before.

The remainder of this paper is structured as follows. Section 2 describes three virtualization techniques as related work which achieve multiple OSes on a single machine. Section 3 explains overview of SIGMA system and detail of its implementation. Section 4 shows evaluations in point of the performance and engineering cost. Section 5 discusses use cases of SIGMA system. Finally, Section 6 summarizes our research.

2 Related Work - Virtualization

The essence of the virtualization technology is a VMM which changes a single hardware interface to multiple interfaces. The interface is duplication of real hardware and equips all instructions (both privileged and unprivileged) of processors and hardware resources (memory and I/O devices, etc...) [3]. Such interface is called a VM, which is the virtualized hardware environment created by a VMM. The VM is controlled by the VMM and provides the same environment as the original hardware. Basically, a program runs on a VM should behave as it runs on an original hardware except some exceptions such as differences caused by the availability of system resources and timing dependencies [14].

Generally, virtualization technologies are categorized into three approaches: full-virtualization, para-virtualization, pre-virtualization.

2.1 Full-virtualization

Full-virtualization is achieved by complete emulation of all instructions and hardware. The most significant advantage of full-virtualization is that almost all OSes

can be run on its VM without requiring any modification to them. In addition, since it emulates hardware completely, the approach does not degrade the reliability of OSes run on the VM. On the other hand, full-virtualization has performance problems. The VMMs developed for full-virtualization are VMware [15][16][19], Virtual PC, real hardware.

2.2 Para-virtualization

Para-virtualization is achieved by modifying a guest OS for a specific hypervisor. The performance of para-virtualization is superior to that of full-virtualization because the OS is optimized for the hypervisor. Though, such modification extinguishes some advantages achieved in full-virtualization. For a examples, a para-virtualized OS can't run on different hypervisors because the OS is created for only a single hypervisor. The examples of para-virtualization hypervisors and its guest OSes are Xen and XenoLinux [2], L4 microkernel and L4Linux [4].

Microkernel-based para-virtualization: A microkernel is one of kernels which has only core functions such as interrupt management, process management, an interprocess communication (IPC). The other functions are implemented as user process modules outside of the kernel. The kernel structure is simple and easy to manage. On the other hand, an IPC is generated very frequently to communicate with the modules and it leads context switches. The performance degradation of a microkernel is mainly caused from those switches. Although some approaches has been proposed to decrease those overheads for some architecture such as arm and x86 [12][17][20], the performance problems in microkernel approaches are the acute issues.

L4Linux is one of the para-virtualized Linux which is ported to run on a L4 microkernel. L4Linux is executed with other servers concurrently in the unprivileged mode. Wombat [9] is also one of the para-virtualized Linux which is ported to run on L4/Iguana system. L4/Iguana consists of NICTA::L4-embedded [8] and Iguana [5]. NICTA::L4-embedded is one of the L4 microkernel modified for embedded systems. Iguana is basic software provides OS services such as memory protection mechanisms and device driver frameworks for embedded systems. NICTA::L4-embedded, Wombat and Iguana have been developed at National ICT Australia (NICTA).

2.3 Pre-virtualization

Pre-virtualization is achieved by modifying assembler codes of a guest OS [10]. The modification consists of multi-phase process called afterburning. Since the afterburning is processed automatically, the engineering cost is relatively small. Pre-virtualization has both benefits of full-virtualization and para-virtualization, the low engineering cost and the high performance. The pre-virtualized OS supports multiple hypervisors such as Xen and L4 [18]. At present, a single compiled image of a pre-virtualized OS can be run on multiple types of hypervisor. Users can select hypervisors for any purpose [11].

3 Design and Implementation

This section explains the design and implementation of SIGMA system, which provides multiple OSes environment especially for multiprocessor architecture. Although SIGMA system looks like a virtual machine monitor in terms of multiple OSes environment, it doesn't provide virtualized hardware interfaces to OSes running on it. An OS in SIGMA system runs on a physical hardware directly. The principal mechanisms to achieve SIGMA system are an interprocessor interrupt (IPI) [6] and an interOS communication (IOSC). The system is implemented on IA-32 architecture.

3.1 Approach and Overview

SIGMA system assigns each processor to a guest OS so that the OS can use all of the CPU resources exclusively. Consequently, this makes SIGMA system simple: it doesn't require a resource management mechanism such as the one implemented in a VMM. OSes of SIGMA system treat those operations by themselves. Those are significant advantages. The first reason is that the OSes hardly require to be modified because the OSes act as if they run on single normal hardware. The second is that the OS in SIGMA system can run in the privileged mode. That is, the OS can handle most of the privileged instructions. As a result, SIGMA system shows as high performance as a native system.

In the Intel architecture, one of the processor is selected as the bootstrap processor (BSP) and others are the application processors (APs) at system initialization by the system hardware [7]. After configuring the hardware, the BSP starts the APs. We define that an OS runs on the BSP is a host OS, and that an OS runs on the AP is a guest OS in SIGMA system. Especially, we call the guest Linux, SIGMA Linux, which is little modified its source codes for SIGMA system. In SIGMA system, NICTA::L4-embedded, Iguana and Wombat are run on the BSP, and SIGMA Linux is run on the AP. The architecture of the system is illustrated in Figure 1.

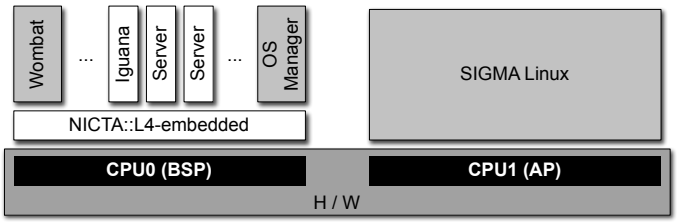

Fig. 1. The structure of SIGMA system

3.2 Boot Sequence of SIGMA Linux

All modules of SIGMA system including a SIGMA Linux kernel are attached to the memory at initial boot time. The OS manager that is one of the modules running on the host OS takes in charge of starting guest OSes. Before SIGMA Linux boots, the OS manager acquires the kernel information from L4. L4 has modules information such as the address in the memory, the image size and other parameters. In addition, to define the memory regions which are available from the guest OS, the OS manager creates new e820 memory maps. The detail of the memory management will be described in next paragraph. The OS manager also writes the information which SIGMA Linux refers such as kernel parameters to the specified memory address. Finally, OS manager sends startup IPI to the AP and SIGMA Linux starts booting. It seems that SIGMA Linux is booted by the general bootloader such as grub on the standalone system. For the guest OS, the OS manager is something like grub.

3.3 Memory Management

Although a normal Linux kernel acquires a memory map through a bios call to identify free and reserved regions in the main memory, a Linux kernel running on SIGMA system obtains the memory map from the OS manager.

Before booting SIGMA Linux, the OS manager executes the program which invokes the e820 function on the AP and acquires the e820 memory map. The e820 function is one of the bios service to check memory size and memory maps. Then, the OS manager modifies the memory map to create memory spaces for SIGMA Linux. The modification is based on the pre-defined memory regions which can be available for SIGMA Linux. After the modification, the OS manager overwrites the modified map to the memory and boots SIGMA Linux. Although the original native Linux includes the instructions which call the e820 function, those are removed from SIGMA Linux to prevent the modified memory map from being overwritten again. Finally, SIGMA Linux reads the modified memory maps and starts booting within the restricted memory areas. This approach is faced with a security problem that SIGMA Linux can ignore and even change this memory restriction. The problem is discussed in Section 5.

3.4 InterOS Communication

The IOSC is a mechanism to communicate between OSes run on the different processors in SIGMA system. The IOSC is similar in mechanism to an interprocess communication (IPC) of a general OS. The IOSC is processed by following steps.

(a) A source OS writes a message to a shared memory region.
(b) The source OS sends an IPI to destination processors.
(c) The destination OS receives the IPI and reads the message.

If the IOSC is called frequently, the system performance can be affected. This resembles IPCs of the microkernel-based system. Though, the IOSC hardly affects the performance of the system because the opportunity to use the IOSC is limited and the frequency is too low as compared to that of IPC in the microkernel-based system.

3.5 Device Management

Device management is mainly achieved by the host OS. Although both the host and guest OSes can use the device directly in fact, if some OSes use the same device at the same time, this device access becomes controversial collision. SIGMA system avoids these collision by restricting the device access to the host OS.

Device drivers are implemented in the device servers of the host OS. A device access of a process in the host OS is performed by just calling the device server. On the other hand, when a process of a guest OS tries to use a device, the process has to send a request to the device server of the host OS by calling IOSC service. This process is performed by a stub driver of the guest OS, called a pseudo device driver. Since the interface of the pseudo device driver is same as common device drivers, the process can use the devices without knowing the location of the device and its driver.

4 Performance Evaluation

This section describes the evaluation of our prototype system. We ran a collection of benchmark programs to evaluate the performance of the guest OS. We also examined the collision on shared memory between the host OS and the guest OS. All experiments described in this paper were performed on a PC with a 2.4GHz Intel Core2 Duo with 32KB L1 cache for each core, 4MB shared L2 cache, 512MB of RAM, and an ATA disk drive. We used Linux kernel version 2.6.12, and 2.6.16 for Xen.

4.1 LMbench

We used the LMbench [13] to measure their performance. LMbench is the micro benchmark software to measure the performance of Linux. LMbench measures the performance of system calls, context switches, latencies of communications and so on. Since LMbench uses regular Linux system calls, it can be suitable to compare many Linux systems.

SIGMA Linux is compared to three systems: vanilla Linux kernel running on a single core (Native Linux), vanilla Linux kernel running on two cores (Native SMP Linux), and modified Linux kernel running on Xen virtual machine monitor (XenoLinux).

Figure 2 shows the cost of the interactions between a process and the guest kernel. The benchmark examines Linux system call and signal handling performance. Any of the items of SIGMA Linux is the same as that of native Linux,

while XenoLinux takes longer time than native Linux in most of the cases. Some experiments of SIGMA Linux made better results than native Linux. This can be caused by the small difference of the environment: SIGMA Linux ignores some interrupts, but native Linux doesn't.

We evaluated the cost of a process context switch in the same set (Figure 3). The cost of a context switch on SIGMA Linux is also almost the same as that of the native Linux and less than that of the native SMP Linux and XenoLinux in any case.

The last benchmark evaluates the cost of the file operations (Figure 4). Xen is as fast as native Linux and SIGMA Linux in file create/delete operations. On the other hand, native SMP Linux is slower than the others. The result shows that SIGMA Linux performs as well as native Linux. It means that there are very few overheads in SIGMA Linux comparing to native Linux.

From the all results, SIGMA Linux presented as the almost same performance as native Linux. In addition, compared to native SMP Linux and XenoLinux, SIGMA Linux shows much better performance than them.

If multiple threads are appropriately assigned to processors and there are no dependencies between the threads, native SMP Linux can achieve efficient processing. Although it may be suitable for math calculation fields and image processing to introduce native SMP Linux, it is not always efficient way for general purpose use. To create a program which is optimized to parallel processing, a special compiler may be required such as automatic parallelizer and it may be also required to use a parallel programming language such as Fortress [1]. As described above, native SMP Linux often shows slower performance than native Linux on a single processor. There are some possible reasons. The first is lock contention: if one processor has already acquired the lock of the critical sections, the other processors have to wait until it will be unlocked to access the regions. In this case, the advantage of parallel execution cannot be made the best use of. The second is the memory access collision issues, which will be described in 4.2. The third is cache inconsistency. In a multiprocessor system, when one processor changes a page table or page directory entry, the change must be propagated to all the other processors to keep the cache consistency. This process is commonly referred to as TLB shootdown and performed by IPI [7]. TLB shootdown in IA-32 architecture must guarantee either of the following two conditions: different TLB mappings are not used on different processors during updating the page table, the OS is prepared to deal with the case where processors use the stale mapping during the table update. A processor can be stalled or must invalidate the modified TLB entries in order to keep the TLB consistency. As a result, this may degrade the system performance.

XenoLinux shows more than two times slower than SIGMA Linux and native Linux at worst. The most substantial element is the emulation of privileged instructions handled by Xen hypervisor. Many of IA-32 instructions related to memory and interrupt and I/O management must be performed in the privileged mode. To perform those operations, XenoLinux invokes hypervisor, which actually handles such instructions. Such invocations are called hypercalls. The

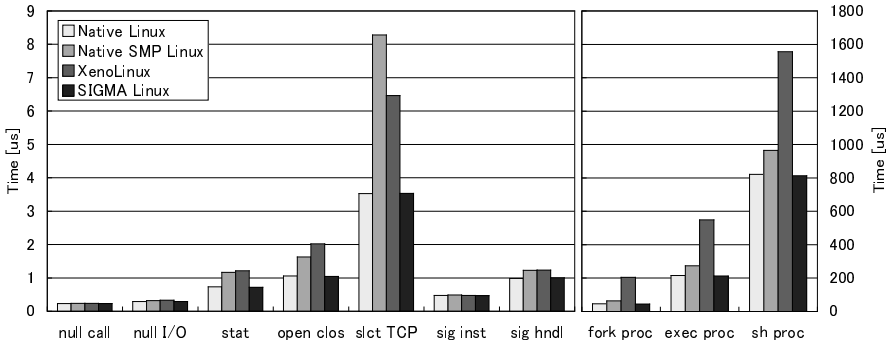

Fig. 2. LMbench: Processes - time in microseconds

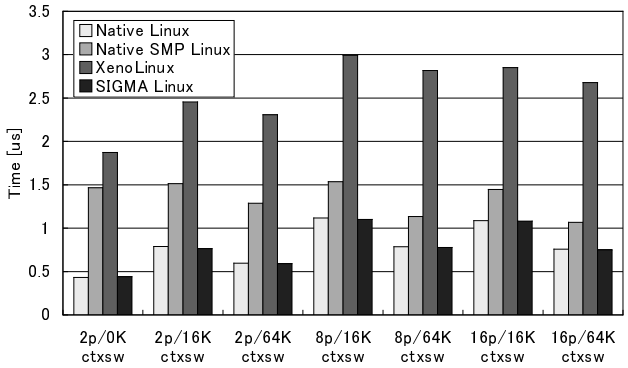

Fig. 3. LMbench: Context switching - time in microseconds

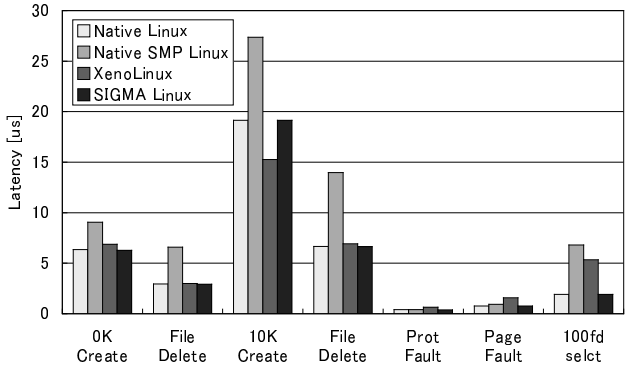

Fig. 4. LMbench: File & VM system latencies in microseconds

hypercall is very expensive processing. Although some optimizations are implemented into Xen to reduce the cost of the hypercall, for a example, some hypercalls are handled together, it can't be enough to improve the performance degradation substantially.

On the other hand, SIGMA Linux shows as fast performance as native Linux. The most significant advantage of avoiding performance degradation is that the guest OS can handle the privileged instructions directly. The configurations of page tables and interrupt vector tables, enabling/disabling interrupts, device controls require privileged instructions and those are usually performed by emulations in general virtualization approaches. Though, SIGMA Linux hardly requires such emulations except in using device drivers run on the other OSes. In addition, the VMM of general virtualization manages the guest OSes and usually handles scheduling of the guest OSes. In other words, the VMM must decide one of the guest OS to run and save and restore all OS contexts. This context switches between OSes are similar to that of processes in general OS and may degrade system performance. In SIGMA system, multiple OSes run concurrently and scheduling of OSes is not required at all. The superiority of SIGMA Linux in performance aspects is based on the architecture of SIGMA Linux which is almost same as that of native Linux.

4.2 Memory Access Collision

Memory accessing can conflict at a memory bus in shared memory multiprocessor architecture. The conflict causes the performance degradation even in SIGMA system. When more than two OSes access to the memory concurrently, while one OS performs memory related operations, the others can be waited for it to become available. We call such conditions memory access collisions and the performance may become slow in such cases. Fortunately, many kinds of multiprocessors have cache memory in each processor and a processor can access cache memory much faster than memory. If a process accesses data exists in cache memory, the read/write operation becomes faster. On the other hand, if the data isn't in cache memory, the process has to access main memory. Such operations are expensive compared to the cache memory. Thus, the results of memory latencies depend on whether the target data is in cache or not. We measured the latencies with memory access collision in five different conditions described in Table 1. The main purpose of the evaluation is to survey how much affection there are from activities of other OSes which share same memory bus.

The five conditions are categorized by cache availability and memory stress. "No Cache" and "Both No Cache" condition mean that caches in each processor are disabled. In "Stress", to generate loading condition, the program which only repeat read and write to memory is run on the host OS, Wombat. In that case, the memory access of the guest OS in "No Cache" condition can be sometimes affected. In "No Cache with Stress" condition, the delay is more increased because the many access from the host OS occurs. In "Both No Cache" and "Both No Cache with Stress" condition, the memory collision is assumed to be much more increased because all processors try to use the memory bus in reading or

Table 1. The measurements condition of memory access collision in SIGMA system

Condition	Host OS		Guest OS
	Cache	Stress	Cache
Normal	enabled	no	enabled
No Cache	no	no	enabled
No Cache with Stress	no	stressed	enabled
Both No Cache	no	no	no
Both No Cache with Stress	no	stressed	no

writing to the memory. The evaluations are performed on the guest OS and we used LMbench as an evaluation tool. The results are illustrated in Figure 5.

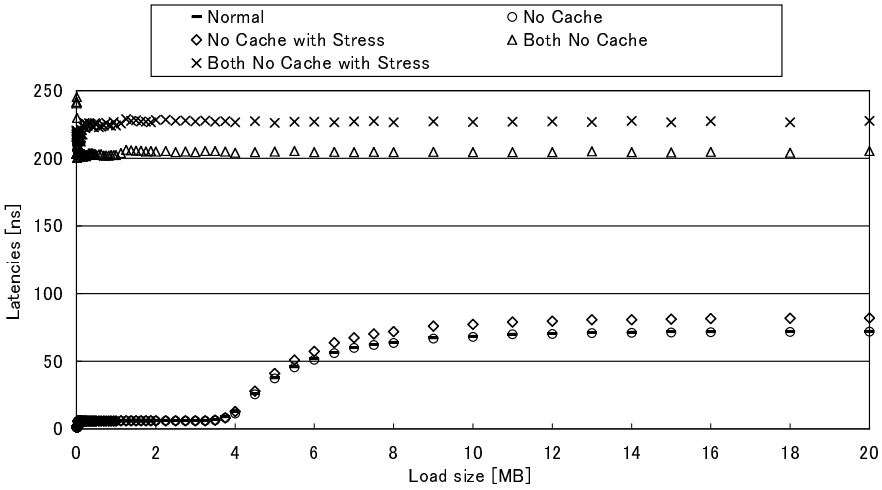

Fig. 5. LMbench: lat_mem_rd: Memory read latencies in the guest OS with memory collisions

Figure 5 shows that, in both cache enabled and disabled configurations, the performance of the guest OS in loading condition degrades ten to fifteen percents compared to that in unloading. In addition, when loading size is below four megabytes, the latencies of "No Cache" and even "No Cache with Stress" conditions are same as the "Normal". That is, no performance degradation occurs below four megabytes when the caches are enabled on the target processor.

In general IA-32 architecture, processors and memory are connected by the 32-bit system bus, and the bus is shared by all processors on the system. Therefore, if multiple processors try to access the memory, the memory access may conflict. In the architecture, memory reading/writing is performed in accordance

with memory ordering model of the processor, which is the order in which the processor issues reads and writes through the system bus to system memory. For a example, in write ordering of multiprocessor system, although each processor is guaranteed to perform writes in program order, writes from all processors are not guaranteed to occur in a particular order [7]. Therefore, if the system memory becomes loaded condition, multiple processors may access to the memory at the same time and actually waiting time can be increased for each processor. As a result, those conflicts caused such delay shown in Figure 5.

On the other hand, the reason why such delay didn't occur in loading four megabytes or less is related to L2 cache size of the processor. If the loading data is in the cache, the processor need not to access the memory and the accessing does not affected by the memory ordering. Therefore, if the each processor has more caches, the possibility of the cache hit increases and it minimizes the delay by the memory access competition, as a result.

4.3 Engineering Cost

We compared the number of modified lines of each guest OS to measure the development cost quantitatively. One of the principle purpose of SIGMA system is to minimize the engineering cost of the guest OS as small as possible. The results are shown in Table 2.

Table 2. The engineering cost of the guest OSes

System	Modified Lines
SIGMA Linux	25
XenoLinux	1441
L4Linux	6500

Table 2 indicates that the modification cost of SIGMA Linux is much smaller than that of XenoLinux and L4Linux. A significant reason is that SIGMA Linux has the almost same structure as native Linux. That is, SIGMA Linux can execute the instructions that control the hardware directly, and such instructions need not to be substituted with emulated codes and hypervisor calls. The required modifications in SIGMA Linux are the memory related issues described in 3.3 and interrupt configurations and hardware checking parts.

5 Discussion

5.1 Limited Memory Protection

SIGMA system can't support memory protection among host and guest OSes because those OSes run in the privileged mode. This means that each OS can

change memory mapping to the MMU of each processor and invade the memory regions where the other OSes manage. Since a VMM of virtualization technologies emulates updating memory mappings, the guest OS can be restricted not to update them by itself to preventing such illegal accessing. On the other hand, SIGMA system doesn't have any methods to control this kinds of illegal access because all OSes run in the privileged mode.

5.2 Estimated Use Cases

The advantages of the SIGMA system are that multiple OSes can run on it concurrently and their performance is as fast as native Linux. The main disadvantage is that SIGMA system has a security problem that its OSes can't be protected each other. Then, we will introduce two systems considered for both the advantages and the disadvantages.

For an Embedded System If an embedded system is once shipped as a product, it is difficult to update its software such as OS and applications. Generally speaking, software of the embedded system is reliable because it is thoroughly verified. Therefore, even if memory protection is not supported sufficiently like SIGMA system, it is possible to introduce SIGMA system to some embedded fields. If there are bugs in OSes or device drivers and the system becomes unstable or crashes at worst, it is also critical condition for existing embedded systems. In other words, even general embedded OS can't ensure that the system continues to work properly in such condition. It is practical to adopt SIGMA system for embedded systems as long as their software is tested sufficiently. The advantages to introduce the SIGMA system to embedded systems are following: inexpensive general-purpose multiprocessors are available in the system, the system makes it easy to reuse the hardware and software by introducing general-purpose hardware platforms.

For a Server System When SIGMA system is used as a server, it is possible to achieve secure and high performance server environment by being incorporated with secure hypervisors.

As illustrated in Figure 6, a secure hypervisor and its guest OS are performed on some processors. The OS has network interfaces and communicates with the other computers. The other OSes which don't depend on the hypervisor run on other processors. Those guest OSes can handle performance-oriented applications without accessing networks.

The OS runs on the hypervisor performs in the unprivileged mode. The OS can never affect the other OSes in the system because the OS is completely managed by the hypervisor. Therefore, it is suitable to run unreliable applications or networking applications on the OS which are always faced with the attack from malicious users. That is, even if the OS are accessed illegally by hitting its bugs or the administrator account of the OS is taken over, the OS can't access the other OSes beyond its own region.

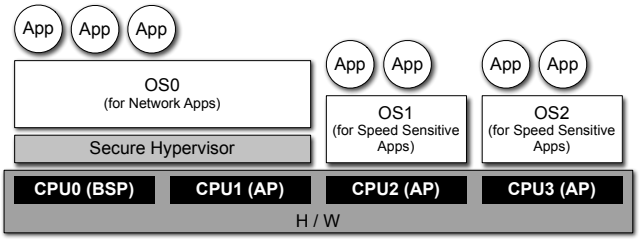

Fig. 6. One of estimated use cases - for a server system

The guest OSes which run on the hardware directly without any hypervisors, OS1 and OS2 in Figure 6, show fast performance, though they have security problems. In other words, such OSes can spoil the other OSes or the hypervisor. Therefore, unreliable and external applications should not be executed on such OSes and it is desirable to reboot the guest OSes periodically to keep them stable. Some applications estimated to be executed on the guest OSes are relatively heavy numeric calculation programs and database management programs and so on. It is possible for an OS runs on the hypervisor to ask the guest OSes to execute such heavy applications by using IOSCs.

As described above, combing SIGMA system with a secure hypervisor which are used in virtualization technologies, both software estimated to be used in reliable condition and performance-sensitive software can be available on the system.

6 Conclusion

We proposed asymmetric multiple OSes environment named SIGMA system on symmetric multiprocessors. Asymmetric means that there are two types of OS run on different processors: L4 microkernel and their servers including OS manager are performed as a host OS, and modified Linux named SIGMA Linux is performed as a guest OS. SIGMA system can avoid performance degradation of SIGMA Linux because SIGMA Linux can control hardware directly without any virtualization.

The evaluation clearly showed that SIGMA system performed much faster than the virtualization techniques such as Xen and almost same performance as native Linux. SIGMA system can be achieved with minimum overheads because no emulations are required to handle privileged processing. We also measured the latency of SIGMA Linux in memory loading condition. The results showed that SIGMA Linux was little affected in the condition. In addition, its engineering cost is much smaller than that of other para-virtualization approaches.

Consequently, the experiments proved that SIGMA system was a practical approach for a performance and engineering cost sensitive system. We believe that SIGMA system is adoptable to a server system and an embedded system.

References

1. E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification Version 1.0 beta*. Sun Microsystems, Inc., March 2007.
2. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the 19th ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.
3. R. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, 1974.
4. H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The Performance of Microkernel-Based Systems. In *SOSP '97: Proceedings of the 16th ACM symposium on Operating systems principles*, pages 66–77, New York, NY, USA, 1997. ACM Press.
5. G. Heiser. *Iguana User Manual*, 4 2005.
6. Intel Corporation. *MultiProcessor Specification Version 1.4*, May 1997.
7. Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, June 2006.
8. I. Kuz. *L4 User Manual NICTA L4-embedded API*, October 2005.
9. B. Leslie, C. van Schaik, and G. Heiser. Wombat: A Portable User-Mode Linux for Embedded Systems. In *Proceedings of the 6th Linux.Conf.Au, Canberra*, 2005.
10. J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie, and G. Heiser. Pre-Virtualization: Slashing the Cost of Virtualization. Technical Report 2005-30, Fakultät für Informatik, Universität Karlsruhe (TH), Nov. 2005.
11. J. LeVasseur, V. Uhlig, B. Leslie, M. Chapman, and G. Heiser. Pre-Virtualization: Uniting Two Worlds, Oct. 23–26 2005.
12. J. Liedtke. Improved Address-Space Switching on Pentium Processors by Transparently Multiplexing User Address Spaces. Technical Report 933, GMD - German National Research Center for Information Technology, Sept. 1995.
13. L. W. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.
14. G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.
15. M. Rosenblum and T. Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *Computer*, 38(5):39–47, 2005.
16. J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. *USENIX Annual Technical Conference*, pages 1–14, 2001.
17. V. Uhlig, U. Dannowski, E. Skoglund, A. Haeberlen, and G. Heiser. Performance of Address-Space Multiplexing on the Pentium. Interner Bericht 2002-01, Fakultät für Informatik, Universität Karlsruhe, 2002.
18. University of Karlsruhe Germany and University of New South Wales and National ICT Australia. Afterburning and the Accomplishment of Virtualization, April 2005.
19. C. Waldspurger. Memory Resource Management in VMware ESX Server. *ACM SIGOPS Operating Systems Review*, 36(si):181, 2002.
20. A. Wiggins, H. Tuch, V. Uhlig, and G. Heiser. Implementation of Fast Address-Space Switching and TLB Sharing on the StrongARM Processor. In *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference*, Aizu-Wakamatsu City, Japan, Sept. 23–26 2003.