A Virtual Machine Monitor for Providing a Single System Image

Kenji Kaneda
University of Tokyo
kaneda@yl.is.s.u-tokyo.ac.jp

Yoshihiro Oyama
University of Tokyo
oyama@yl.is.s.u-tokyo.ac.jp

Akinori Yonezawa
University of Tokyo
yonezawa@yl.is.s.u-tokyo.ac.jp

Abstract

We have designed and implemented a virtual machine monitor that virtualizes a shared-memory multi-processor machine on a commodity cluster. This functionality greatly simplifies utilization of commodity clusters. For example, it enables parallel applications for shared-memory multi-processor systems to run on clusters without any change of the applications. Moreover, commodity operating systems that support multi-processors (e.g., Linux) can be installed in a virtual machine with a small amount of modification. We built a virtual 8-way multi-processor machine on eight physical machines. We ran parallel coarse-grain tasks on Linux installed in the virtual machine and measured the execution time. The experimental result demonstrates the feasibility of our approach.

1. Introduction

Due to the recent increase in the performance/price ratio of PCs, there is rapidly expanding interest in the use of computing clusters composed of commodity computers. While the commodity clusters provide scaling from the small (less than 64 nodes) to the large (approaching 10,000 nodes), small clusters are gaining widespread use, particularly at the workgroup and departmental levels.

A major problem for utilizing such small commodity clusters has been the complexity of resource management. Without global (cluster-wide) mechanisms for resource allocation and sharing, it is difficult to efficiently utilize resources such as processors, memory, and disks.

To overcome this problem, we propose a method for providing a single system image (SSI) on top of a cluster. While various systems (e.g., SCore [23], Condor [17]) have been proposed to provide a SSI, we especially focus on a technique for achieving a SSI with *hardware virtualization*. More specifically, we designed and implemented a virtual machine monitor (VMM) called *Virtual Multiprocessor*. Like existing VMMs [4, 6, 8, 10, 11, 18, 27, 29, 32], Virtual Multiprocessor takes complete control of the machine hardware and creates virtual machines, each of which behaves like a complete physical machine that can run its own operating system. In contrast to the existing VMMs, Virtual Multiprocessor virtualizes a shared-memory multi-processor machine on a commodity cluster. For example, it gives a user the illusion of an N-way multi-processor machine on top of a collection of N single-processor

machines. Inside the virtual machine, the user installs an operating system that supports multiprocessor machines and executes parallel programs on the operating system.

Our approach to achieving a SSI has three advantages over existing approaches. First, a wide variety of parallel applications for shared-memory multi-processor systems can run in a virtual machine built on the cluster without any changes of the applications. Especially, a user can write parameter sweep applications or parallel tasks that have DAG dependency between them using familiar languages and tools designed for shared-memory systems (e.g., parallel `make`, shell script) instead of distributed programming languages such as MPI [25].

Second, in addition to parallel applications, execution of multiple sequential applications gains benefit from our approach. By installing a commodity operating system that supports multiprocessors (e.g., Linux) in a virtual machine, the user can manage distributed resources with a familiar interface such as Linux process management. If the user forks multiple processes on Linux running inside the virtual machine, these processes are automatically allocated on the virtual machine's processors by the scheduling mechanism of Linux. The processes are then allocated on the physical machines' processor which the virtual machine's processors are mapped onto.

Third, resource encapsulation with our VMM gives solutions for security and reliability. For example, suppose a cluster is used for server hosting. Since a VMM provides strong isolation between virtual machines and physical machines, the administrator of a VMM can give full control of the virtualized hardware to the users of the virtual machines, without exposing critical resources to danger. This functionality of VMMs greatly supports to achieve secure and convenient virtual hosting [30, 32]. In addition, a snapshot/resume mechanism of virtual machines enables a system to reduce the effects of system crashes and breakins [21, 29, 31].

The current implementation of the VMM is designed for the IA-32 architecture. The VMM virtualizes processors, shared memory, and I/O devices as follows. To virtualize processors, the VMM achieves para-virtualization of the IA-32 instruction set architecture (ISA) [4, 10]. A guest operating system is statically modified to run optimally on a virtual machine. To virtualize shared-memory, the VMM uses a mechanism similar to software distributed shared memory. The VMM implements the consistency protocol of the shared memory with the virtual memory page protection mechanism of physical machines. To virtualize I/O devices, the VMM prepares a central server that keeps track of the states of all the devices. The VMM communicates with the server whenever a virtual processor issues an I/O operation.

We conducted several experiments to demonstrate the feasibility of our approach from a performance perspective. We built a virtual 8-way multi-processor machine on eight physical machines. We ran eight processes that calculate a fibonacci number simultaneously on Linux installed in the virtual machine and measured the execution time. The execution of the program on a virtual 8-way multi-processor machine is about 6.6 times faster than on both virtual and physical 1-way processor machines. These results indicate that applications that do not require a large amount of the VMM interventions (e.g., do not access I/O devices very frequently) achieves good performance.

The remainder of this paper is organized as follows. Section 2 presents the overview of Virtual Multiprocessor. Section 3 describes the implementation of the virtualization of hardware resources. Section 4 gives the details of memory consistency algorithm. Section 5 presents performance measurements. Section 6 discusses limitations of our system and proposes several solutions for overcoming the limitations. Section 7 discusses related work. The final section summarizes the paper.

2. Overview of Virtual Multiprocessor

This section presents the overview of Virtual Multiprocessor. First, we describe the basic design decisions. Next, we explain how Virtual Multiprocessor maps virtual resources to physical resources.

2.1. Functionality of Virtual Machines

Functionality of virtual machines built by our VMM is summarized as follows:

- An interface provided by the virtual machines is not at the ABI level but at the ISA level. The virtual machines provide a complete system environment that supports an operating system along with user processes.
- Both the virtual machines and underlying physical machines are targeted at the IA-32 architecture.
- The VMM achieves partial virtualization of an underlying machine (i.e., para-virtualization [4, 32]) as opposed to full virtualization [29].

Due to para-virtualization, the ISA of the virtual machines is similar but not identical to that of underlying hardware. This improves performance, though the kernel of operating systems running inside the virtual machine requires a small amount of modification. The technique of our VMM for modifying operating systems is similar to that of LilyVM [10]. We describe the details of the technique in Section 3.

2.2. Mapping of Hardware Resources

Virtual Multiprocessor maps hardware resources (processors, memory, and I/O devices) of a virtual machine onto those of physical machines to virtualize a shared-memory multi-processor machine. As shown in Figure 1, the resources are mapped in a following manner:

Processors Virtual processors are basically mapped onto physical processors in a one-to-one fashion. N individual processors of a virtual machine are respectively mapped onto a processor of N different physical machines.

Memory A virtual machine's shared memory available to any of the virtual processors is mapped onto a portion of physical machines' memory. Each physical machine needs to reserve M MB of memory to virtualize M MB of the shared memory.

I/O devices I/O devices of a virtual machine are mapped onto devices belonging to one of physical machines. For example, a disk image file located at one of physical machines is used as a hard disk image of a virtual machine. A virtual console of a physical machine is used for a serial terminal of a virtual machine.

3. Implementation

This section describes how Virtual Multiprocessor virtualizes the IA-32 architecture: processors, shared memory, and I/O devices. The virtualization of shared memory consists of the virtualization of address space and coherence mechanism. We give just the outline of the virtualization of processors and address space since it is similar to that of a single-processor virtual machine, particularly to LilyVM [10]. The virtualization of memory coherence mechanism and I/O devices is described in more detail because these mechanisms are special for the virtualization of a multi-processor machine.

3.1. Basic Strategy for Virtualizing Hardware

Like LilyVM [10] and FAUmachine [11], our VMM is placed on top of a native operating system running on physical hardware and is implemented solely in user mode with no modification

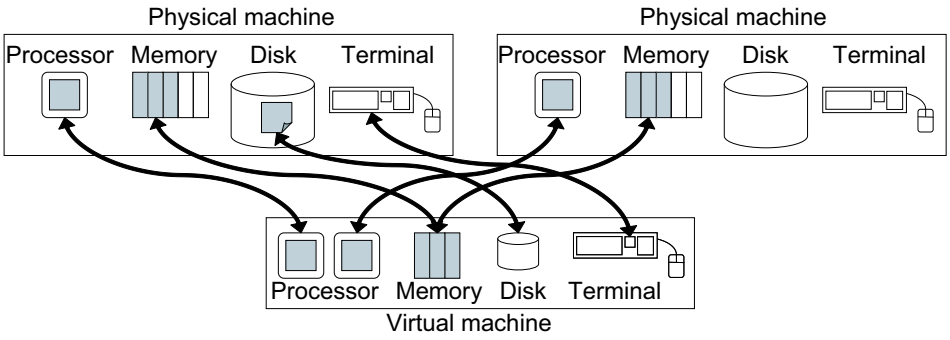

Figure 1. Mapping between a virtual machine and physical machines

to the native operating system. Although this architecture incurs a larger overhead than the architecture where the VMM is placed directly on bare hardware [4, 30, 32], it overcomes several technical and pragmatic hurdles [24]. First, the architecture of our system is useful for the virtualization of the Intel Pentium architecture, which is not naturally virtualizable [20]. Second, by relying upon a native operating system, it allows a virtual machine to support a diversity of peripheral devices with minimal programming effort. Third, it allows an operating system installed inside a virtual machine to co-exist a pre-existing native operating system. Hereafter, we call an operating system running on a virtual machine a *guest* operating system and an operating system running on a physical machine a *host* operating system.

To virtualize hardware resources with no modification to host operating systems, the VMM prepares two user processes for each virtual processor. These user processes are:

VM process The VMM assigns this process to run a guest operating system as one of processors of a virtual machine. The individual VM processes map assigned virtual processors onto physical machines' processor where the VM processes are running.

When the VM process is about to execute an instruction that interferes with the state of an underlying VMM or a host operating system, a signal is generated by a host operating system. For example, the `SIGSEGV` signal is generated when the VM executes a privileged instruction which cannot be executed by a user process.

Monitor process This process supervises the VM process using the `ptrace` system call. The monitor process intercepts execution of the VM process by trapping a signal generated by the VM process. The monitor process then emulates the instruction executed by the VM process by modifying the state of the VM process's registers and memory.

Figure 2 summarizes a basic execution cycle of these processes.

3.2. Processor Virtualization

The virtualization of processors consists of (i) the virtualization of instructions that would interfere with the state of an underlying VMM (or a host operating system) and (ii) the virtualization of interrupts and exceptions.

First, we describe the virtualization of instructions that would interfere with underlying systems. As mentioned in Section 3.1, a large portion of a virtual processor's instructions is executed by a physical machine's processor without VMM intervention. Only instructions that would interfere with an underlying VMM or host operating system are interpreted by the VMM. These instructions that require VMM intervention are called

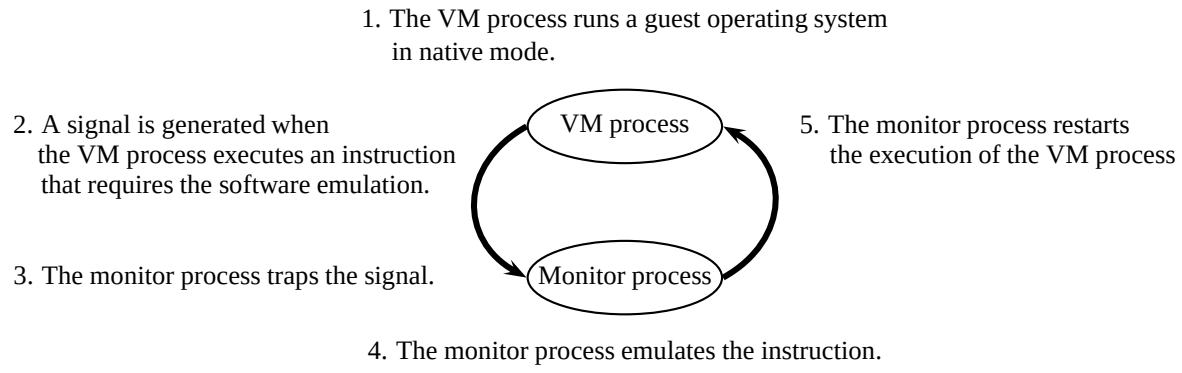

Figure 2. A basic execution cycle of the VM process and the monitor process

sensitive instructions. For example, instructions that access IA-32 system registers such as control register 3 are sensitive.

The sensitive instructions are classified into privileged instructions (e.g., the `lgdt` instruction) and non-privileged instructions (e.g., the `sgdt` instruction) [20]. Execution of the privileged instructions at the most privileged hardware domain will cause a general protection exception, whereas the non-privileged instructions do not cause an exception.

The monitor process traps the execution of privileged instructions and non-privileged instructions in different ways. Trapping of privileged instructions is straightforward. Since a VM process runs in user mode, a monitor process needs only to trap exceptions caused by the execution of privileged instructions. On the other hand, trapping of non-privileged instructions is complex and requires modifications to a guest operating system. More specifically, the kernel code of a guest operating system is modified at compile time in such a way that an illegal instruction is inserted before every non-privileged instruction [10]. By trapping the exception caused by an illegal instruction, the monitor process intercepts non-privileged instruction that follows the illegal instruction in the kernel code.

This technique of the static modification of kernel code has merits and demerits. One of the merits is that numerous operating systems can be

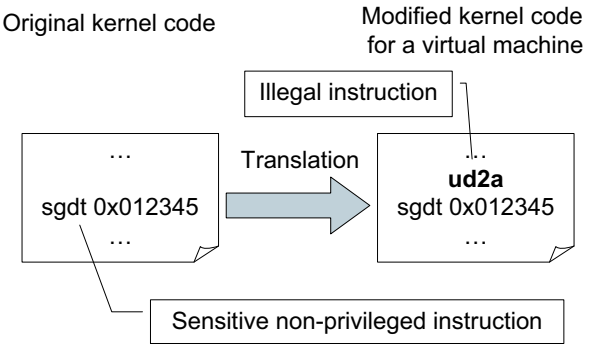

Figure 3. Translation of kernel code with a modified assembler

hosted with small manual implementation costs. A modified assembler inserts illegal instructions automatically at kernel compile time. On the other hand, due to static code modification, the VMM cannot support system-level binaries of which source code is not available. These include binary-only Linux kernel modules and operating systems like Windows, of which source code is not open to public.

Second, we describe the virtualization of interrupts and exceptions. To virtualize interrupts and exceptions, the VMM needs to detect and deliver them. A method for detecting interrupts and exceptions varies depending on how they are generated. For example, when a virtual machine generates an exception that can be seen as a signal generated by the VM process, the monitor process

detects it by trapping the signal with the `ptrace` system call. When the virtual machine generates an interrupt by accessing its APIC, the monitor process detects it by intercepting write access to memory regions which the virtual machine's APIC is mapped onto.

A detected interrupt (or exception) is delivered to an appropriate virtual processor by the monitor process. Basically, the monitor process delivers it to the local VM process. The monitor process makes the VM process enter an interrupt (or exception) handler by looking up descriptor tables. Only when an inter-processor interrupt is generated, the monitor process delivers it to a specified remote VM process with TCP/IP communication. The delivery of external interrupts triggered by I/O devices is described in Section 3.4 in detail.

3.3. Shared Memory Virtualization

The virtualization of a shared memory requires the virtualization of the address space and the memory coherence mechanism.

First, we briefly explain the virtualization of the address space. A guest operating system running inside a virtual machine expects a zero-based physical address space, as provided by real hardware. To implement such address space, the VMM need virtualize both the segmentation mechanism and the paging mechanism. In current implementation, the segmentation mechanism is not fully virtualized. Only minimal mechanism required to host Linux is implemented. Specifically, whereas reading from and writing to a virtual machine's segment registers are implemented, translation from virtual addresses to physical addresses is not fully supported. The base address of every segment must be zero inside a virtual machine.

The virtualization of the paging mechanism is implemented in the following manner. First, an individual VM process reserves a portion of its memory for a virtual machine. Then, the VM processes map their pages onto the reserved memory region by looking up the page directory and the page table of the virtual machine.

More specifically, the VM processes use the `mmap` system call and the `munmap` system call to update the mapping of pages. Since these system calls incur a large overhead, the VM processes delay the system call invocation; the system calls are issued only when the modification to the page directory and the page table need become valid. For example, suppose that a virtual machine modifies its page table so that page p is mapped onto its physical memory. In this case, the VM process delays issuing the `mmap` system call until the process actually accesses to p and the `SIGSEGV` signal is generated. Similarly, the VM process issues the `munmap` system call to release obsolete page mapping only when modification to the page table or page directly becomes valid. For instance, `munmap` is issued to release obsolete mapping when a virtual machine changes the value of the control register 3 or execute the `invlpg` instruction.

The virtual address space that a guest operating system can access is limited. An upper bound of the address space is changed to `0xaffffffff` for the following reasons:

- A memory region with lower bound `0xc0000000` and upper bound `0xffffffff` is used for the kernel address space for a host operating system. A VM process, which runs not in supervisor mode but in user mode, is not allowed to access this region by a host operating system.

- A memory region with lower bound `0xb0000000` and upper bound `0xbfffffff` is reserved for hardware emulation. This region is used for storing information required for the virtualization of hardware such as values of system registers.

For the above reasons, a guest operating system is statically modified such that its kernel

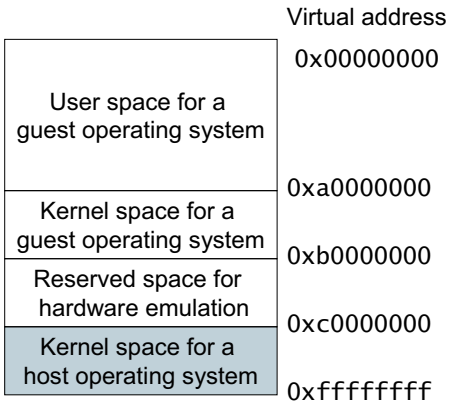

Figure 4. Memory layout of the monitor process

address space does not overlap with the non-accessible regions. A lower bound and an upper bound of the kernel address space are changed to `0xa0000000` and `0xb00000000` respectively (See Figure 4).

Next, we describe the virtualization of memory coherence mechanism. The VMM implements the consistency protocol of the shared memory using the virtual memory page protection mechanism of physical machines. Specifically, the VMM uses the `mprotect` system call to control access to shared pages in such a way that any attempt to perform a restricted access on a shared page generates the `SIGSEGV` signal. Upon trapping this signal, the VMM updates the contents and protection level of the page on the physical machine. The details of the memory sharing mechanism are described in Section 4.

It must be noted that the `SIGSEGV` signals are generated by several reasons, including a page fault exception of a virtual machine and access to a shared page of which privilege is downgraded. Since the way the `SIGSEGV` signals are handled varies depending on reasons the signals are generated, the VMM classifies the signals according to the flow-chart shown in Figure 5.

3.4. I/O Device Virtualization

I/O devices currently supported by the VMM include a hard disk and a serial terminal. The supported access method to these devices include programmed I/O (with `in/out` instructions) and Direct Memory Access (DMA). Access through memory mapped I/O is not currently implemented.

To emulate I/O devices, the VMM prepares one central server that keeps track of the states of all the devices. We call this I/O server. The I/O server communicates with monitor processes to emulate the I/O devices. For example, when a guest operating system tries to read a value from an I/O port with the `in` instruction, the I/O server and a monitor process emulate the instruction as follows. First, the monitor process intercepts the execution of the `in` instruction and sends a request to the I/O server. When receiving the request, the server reads a value from the specified I/O port and sends it to the monitor process. The monitor process then copies the value to the destination operand of the instruction.

To trigger external interrupts generated by the devices, the I/O server checks the state of the devices at regular intervals and tries to find the devices that can trigger an interrupt. If such a device is found, the server delivers an external interrupt of the device to a virtual processor in the following manner. First, the server decides a destination virtual processor to which the interrupt is delivered. In the current implementation, the interrupt is delivered to virtual processor v such that v and the I/O server run on the same host [1]. Second, the I/O server transmits some signal (e.g., `SIGUSR1`) to the VM process corresponding to v to stop its execution. Finally, the monitor process traps the signal and makes the VM process enter an interrupt handler.

[1]The scheduling of destinations with dynamic priority has not been implemented yet.

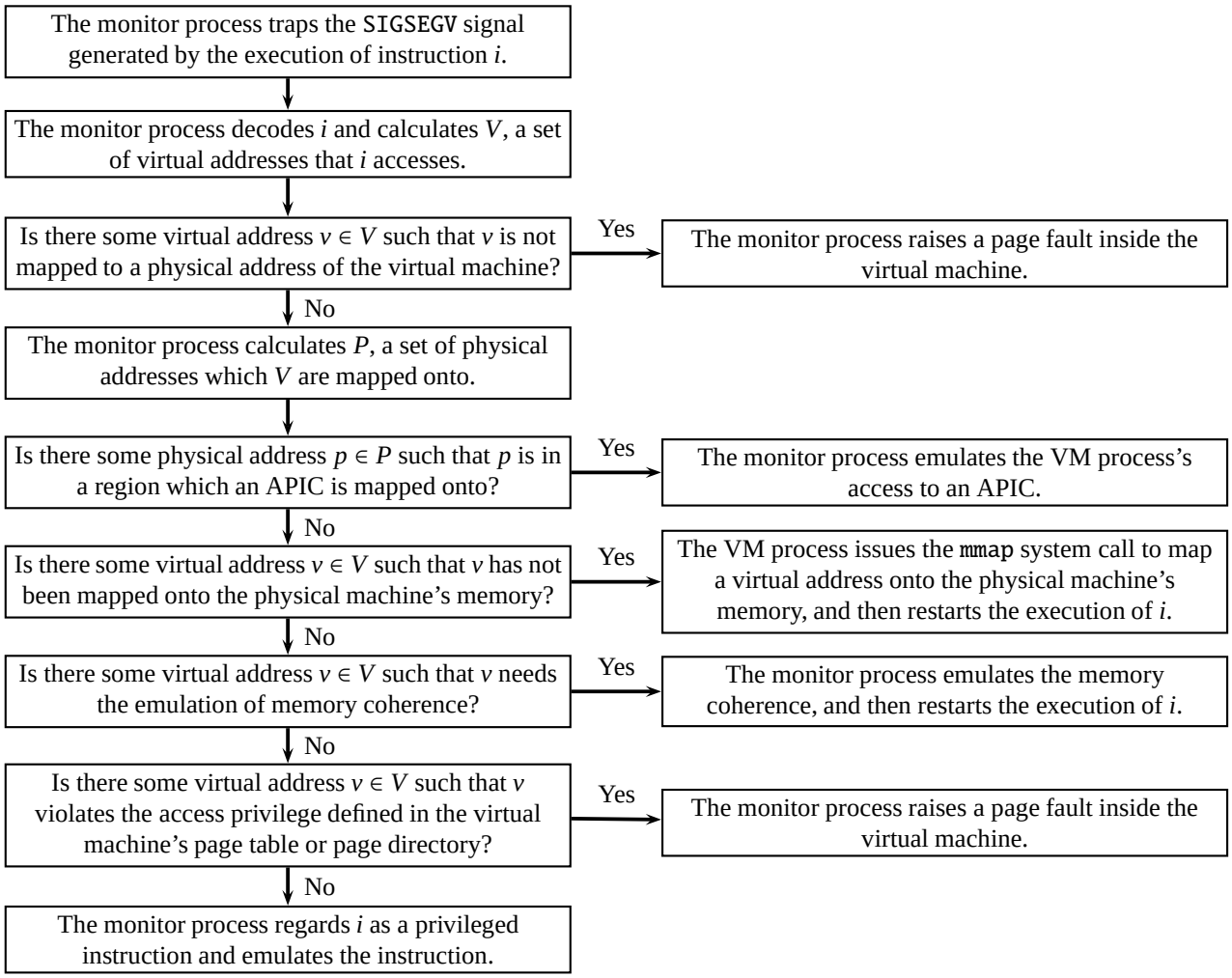

Figure 5. Flow chart of the SIGSEGV signal handling

$proc_i$	:	virtual processor i
M_i	:	message queue of virtual processor i
$pages_i^n$	:	nth page of virtual processor i
$p.state$	:	state of page p (`invalid`, `read_only`, or `read_write`)
$p.content$	:	content of page p
$p.owner$	:	processor that own the latest content of page p
$p.copyset$	:	a collection of processors that have a replica of page p
$p.busy$	:	flag which is `true` while page p is being updated

Figure 6. Variables for algorithm description

4. Memory Consistency Algorithm

This section explains the virtualization of the memory coherence mechanism in detail. First, we describe the IA-32 memory model. The VMM need satisfy this memory model to allow existing programs for the IA-32 architecture to run inside a virtual machine without modification. Then, we present a simple memory consistency algorithm that satisfies the IA-32 memory model. Note that the memory model that our algorithm targets differs from most of existing memory consistency algorithms for distributed shared memory systems designed for other memory models such as release consistency [5, 15].

4.1. IA-32 Memory Model

The IA-32 memory model specifies the order in which processors see updates to memory so that the processors. According to its specification [12], the IA-32 memory model guarantees that the following ordering rules apply in multiprocessor machines:

- Individual processors use the same ordering rules as in a single-processor machine.
- Writes by a single processor are observed in the same order by all processors.
- Writes from the individual processors are *not* ordered with respect to each other.

Added to the above ordering rules, the IA-32 architecture provides several mechanisms for strengthening or weakening the memory ordering model to handle special programming situations. These mechanisms include the I/O instructions, locking instructions, the LOCK prefix, and serializing instructions that force stronger ordering on processors. For instance, `mfence` is one the serializing instructions. It ensures that every load-from-memory and store-to-memory instructions that precede the `mfence` instruction in machine code is globally visible when the `mfence` instruction is issued.

4.2. Algorithm Description

We explain a simple memory consistency algorithm that satisfies the IA-32 memory model. This algorithm is based on a simple sequentially consistent, multiple-reader/single-write protocol used in Ivy [14]. The machine pages of the virtual machine are distributed over nodes such that each node manages a subset of the pages. Figure 6 and Figure 7 describe the algorithm in more detail. Figure 6 shows variables used in the algorithm description. Figure 7 describes actions of virtual processor i. When one of the conditions listed on the left side of the figure holds, a corresponding action listed on the right side of the figure is took.

We plan to optimize the algorithm by relaxing memory consistency as far as the IA-32 memory model can be satisfied. This optimization plan is discussed in Section 6.2.

Guard		Action
$access(i,a,n)$ $\wedge$ $violation(pages_i^n, a)$	$\Longrightarrow$	**begin** stop the execution of the VM process; send $\langle$**fetch**$, n, a, i\rangle$ to $manager(n)$; **end**
$\langle$**fetch**$, n, a, s\rangle \in M_i$ $\wedge$ $pages_i^n.busy =$ `false`	$\Longrightarrow$	**begin** remove $\langle$**fetch**$, n, a, s\rangle$ from M_i; let p be $pages_i^n$; $p.busy :=$ `true`; **match** a **with** `read` $\Rightarrow$ send $\langle$**invalidate**$, n, a, s, p.owner\rangle$ to $p.owner$; `write` $\Rightarrow$ **forall** $x \in p.copyset$ such that $x \neq proc_s \vee x = p.owner$ **do** send $\langle$**invalidate**$, n, a, s, p.owner\rangle$ to x; **end** **end**
$\langle$**invalidate**$, n, a, s, o\rangle \in M_i$	$\Longrightarrow$	**begin** remove $\langle$**invalidate**$, n, a, s, o\rangle$ from M_i; let p be $pages_i^n$; **match** a **with** `read` $\Rightarrow p.state :=$ `read_only`; `write` $\Rightarrow p.state :=$ `invalid`; **end**; **if** $o = proc_i$ **then** send $\langle$**ack**$, n, a, p.content\rangle$ to $proc_s$; **end**
$\langle$**ack**$, n, a, c\rangle \in M_i$	$\Longrightarrow$	**begin** remove $\langle$**ack**$, n, a, c\rangle$ from M_i; let p be $pages_i^n$; $p.content := c$; **match** a **with** `read` $\Rightarrow p.state :=$ `read_only`; `write` $\Rightarrow p.state :=$ `read_write`; **end**; send $\langle$**finish**$, n, a, i\rangle$ to $manager(n)$; restart the execution of the VM process; **end**
$\langle$**finish**$, n, a, s\rangle \in M_i$	$\Longrightarrow$	**begin** remove $\langle$**finish**$, n, a, s\rangle$ from M_i; let p be $pages_i^n$; **match** a **with** `read` $\Rightarrow p.copyset := p.copyset \cup \{ proc_s \}$; `write` $\Rightarrow p.copyset := \{ proc_s \}$; $p.owner := proc_s$; **end**; $p.busy :=$ `false`; **end**

where

$manager(n)$	:	manager of nth page (e.g., $manager(n) = proc_{n \bmod N}$ where the number of processors is N)
$access(i,a,n)$	:	This predicate holds when processor i accesses with a (`read` or `write`) to nth page
$violation(p,a)$	$\equiv$	$p.state =$ `invalid` $\vee$ ($p.state =$ `read_only` $\wedge$ $a =$ `write`)

Figure 7. Simple memory consistency algorithm (for virtual processor i)

Name	Descripton	Execution time (physical)	Execution time (virtual)	Overhead ratio
`fib`	Calculate a fibonacci number	22.6	22.1	0.97
`getpid`	Issue `getpid` $100,000$ times	0.05	18.1	354
`ls`	List information about hundreds of files	0.03	6.64	255
`gcc`	Compile a C program	0.14	0.98	6.81

Table 1. Sequential benchmark programs and their execution time on a physical and a virtual single-processor machine (unit: seconds)

5. Experiments

We implemented a prototype of Virtual Multi-processor and conducted several experiments to demonstrate the feasibility of our system from a performance perspective. This prototype system builds a virtual 8-way multi-processor machine on top of eight physical machines. The virtual machine can host the Linux kernel for SMP and allows various applications (e.g., `gcc`, `make`) to run on Linux.

Specifically, we conducted the following experiments. First, we ran several sequential programs on a virtual single-processor machine to measure the overhead of hardware virtualization except the memory coherence mechanism. The hardware virtualization involves emulation of sensitive instructions, access to I/O devices and so on. Second, we ran parallel coarse-grain tasks on a virtual multi-processor machine to solely measure the overhead of the memory coherence mechanism.

All the experiments were conducted on 2.4 GHz Intel Xeon machines with 2GB RAM, a 1 Gigabit Ethernet NIC. Linux 2.4 was used throughout for both a host operating system and a guest operating system.

5.1. Execution of Sequential Programs on a Virtual Single-processor Machine

We ran several sequential programs on a virtual single-processor machine to measure the overhead of hardware virtualization except the memory coherence mechanism.

Table 1 shows description of benchmark programs and their execution time on a physical machine and a virtual machine. Although the experimental result indicates that overheads incurred by the execution of `getpid`, `ls`, and `gcc` are large, the overheads can be reduced as indicated by the performance of existing IA-32 VMMs (e.g., VMware [29], Xen [4]). Section 6.1 discusses several techniques for reducing the overheads.

5.2. Execution of Parallel Coarse-grain Tasks on a Virtual Multi-processor Machine

We measured the execution time of parallel coarse-grain tasks on a virtual multi-processor machine to evaluate the overhead of the memory coherence mechanism. Specifically, we ran eight processes that calculate a fibonacci number simultaneously on a 1-way, ..., 8-way virtual multi-processor machine built on top of 1, ..., 8 physical machines respectively. The overheads of this program are mainly due to emulation of the following hardware mechanism:

- System calls such as `fork` used for creating processes.
- Access to a hard disk for loading the executable file and shared libraries.
- The memory coherence mechanism (especially required for processes running in kernel mode).

Figure 8 shows the speedup of this program. `fib(n)` denotes the calculation of `n`th fibonacci

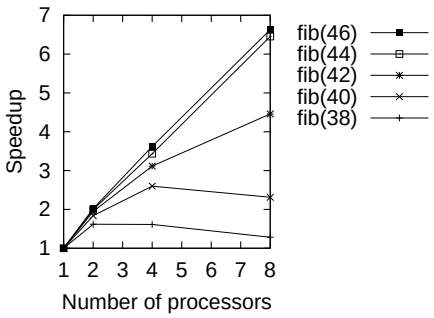

Figure 8. Speedup of parallel fibonacci

# of procs.	Total	Native	Shmem	Misc	Idle
1	180.0	177.8	0.0	2.2	0.0
2	90.3	87.9	1.0	1.1	0.3
4	52.4	43.7	3.0	0.4	5.3
8	27.9	22.1	3.7	0.1	2.0

Table 2. Breakdown of execution time of `fib(44)` (unit: seconds)

number. As shown in this figure, the program achieved better speedup as tasks were coarser. The execution of `fib(46)` on an 8-way multiprocessor machine is about 6.6 times faster than on a 1-way processor machine.

Table 2 gives the breakdown of the execution of `fib(44)`. 'Total' denotes the total execution time of `fib(44)`. 'Native' denotes a time how long the virtual machines ran in native mode. 'Shmem' denotes a time spent for the virtualization of the memory coherence mechanism. 'Misc' denotes a time spent for the hardware virtualization other than the memory coherence mechanism. 'Idle' denotes a time how long the virtual machines executed the 'hlt' instruction. For more than one processor, the table shows the average of the execution times of individual processors. Table 2 indicates that the overhead is mainly caused by the virtualization of the memory coherence mechanism, which becomes larger as the number of processes increases.

We further investigate the overhead incurred by the virtualization of the memory coherence mechanism for `fib(44)`. We measured the distribution of virtual addresses fetched by the monitor processes for memory sharing (Figure 9) and the distribution of times which individual page fetch requests took to complete (Figure 10). Figure 9 indicates that page fetch requests frequently occurred at the beginning and the end of `fib(44)` (in both user and kernel mode). Note that the base address of the kernel space of the guest operating system is changed to `0xa0000000` as mentioned in Section 3.3. Figure 10 shows that some of fetch requests took tens of milliseconds to complete whereas most of page fetches were completed in less than ten milliseconds. These experimental results indicate that the overhead of the parallel coarse-grain tasks is due to frequent page fetches caused by false sharing.

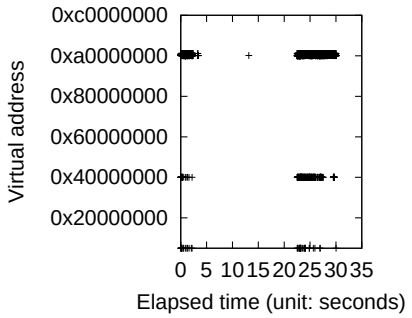

Figure 9. Distribution of virtual addresses fetched by the monitor processes for memory sharing (for `fib(44)`)

6. Discussion

In this Section, we discuss limitations of the current implementation of Virtual Multiprocessor and propose several solutions for overcoming the limitations.

6.1. Optimization of Hardware Virtualization

Currently, our VMM is placed on top of a host operating system and is implemented solely in user mode without any modifications to the host

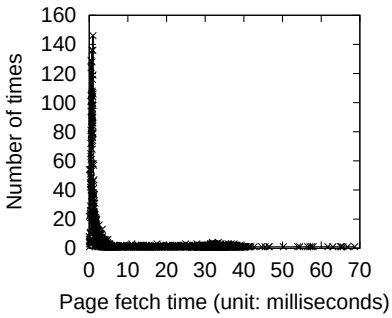

Figure 10. Distribution of times which individual page fetch requests took to complete (for `fib(44)`)

operating system. Although this architecture requires only small amounts of implementation efforts, the virtualization of IA-32 architecture (e.g., issues of system calls, access to I/O devices) incurs a larger overhead as shown in Section 5.1.

To reduce these overheads, we plan to apply existing optimization techniques developed by numerous IA-32 VMMs (e.g., Xen [4], CoVirt [16]) to our VMM. For example, the number of context switches caused by the `ptrace` system call can be reduced by placing the VMM directly on bare hardware like Xen. We also plan to port the VMM to other architectures of which design is more suitable for hardware virtualization [1, 13].

6.2. Optimization of Memory Consistency Algorithm

Since the memory consistency algorithm described in Section 4.2 is simple sequentially consistent, we plan to develop an algorithm that relaxes memory consistency as far as the IA-32 memory model can be satisfied.

An example of optimization techniques that we are planning is to allow multiple nodes to write to the same page simultaneously. The sequentially consistent algorithm does not allow multiple nodes to write to the same page at the same time since writes updates need to become globally visible immediately. In contrast, our optimized algorithm delays write updates until a synchronous instruction or an atomic instruction is issued. Note that this relaxation of memory ordering does not violate the IA-32 memory model according to the specification [12].

6.3. Fault Tolerance

A machine crash is a frequent event in commodity clusters, in which a large number of machines involve. Hence we require that the system can continue to run even if some machines fail. We plan to implement the fault tolerance mechanism using techniques such as the checkpointing/recovery [9] and replication for VMMs [26].

7. Related Work

7.1. Virtual Machine Monitors

Recently several VMMs that build a virtual multi-processor machine have been developed. These VMMs include vNUMA [7], Virtual Iron [28], Disco [6], and VMware ESX Server [30].

vNUMA virtualizes a cc-NUMA machine on top of physical machines with the Itanium architecture. Whereas the memory coherence mechanism of vNUMA invalidates memory pages synchronously, pages are invalidated asynchronously in Virtual Multiprocessor. This asynchronous page update reduces downtime when a guest operating system is not running though the total number of messages required for each page fetch increases.

Virtual Iron [28] builds a virtual multiprocessor machine on top of clusters. The basic mechanism of Virtual Iron is similar to that of our system. A comparison between Virtual Iron and our system has not been made yet since details of Virtual Iron are not public (2005/10/14).

Disco and VMware ESX Server require a physical machine that has an equal or greater number of processors as they are attempting to virtualize.

In contrast, our VMM can build a virtual multiprocessor machine regardless of the number of physical processors and the number of machines on which these processors reside. This functionality of our VMM allows users to harness distributed resources efficiently and transparently.

7.2. Middlewares and Operating Systems for Providing a SSI

Middleware systems for clusters (e.g., SCore [23] and Condor [17]) provide a single software image for high-performance parallel programming environments. However, an interface provided by these systems differs from that of commodity operating systems. In contrast, our system's interface is same as that of commodity operating systems. This functionality greatly simplifies the utilization of distributed resources.

There are several systems (e.g., MOSIX [3] and Kerrighed [19]) that enhance the Linux kernel with cluster computing capabilities. Drawbacks that these systems suffer include large implementation costs for kernel modification and difficulty in supporting numerous operating systems.

7.3. Software Distributed Shared Memory Systems

Shasta [22] and cJVM [2] are software distributed shared memory systems that transparently support a shared address space across a cluster of workstations. Shasta implements the shared address space by transparently rewriting the application executable to intercept loads and stores. cJVM implemented the shared memory space by modifying Java Virtual Machine.

While Shasta and cJVM support only user programs, our system allows an entire operating system for SMP to run on clusters. Furthermore, our system is targeted at the IA-32 architecture whereas Shasta is targeted at the MIPS architecture and cJVM at Java Virtual Machine.

8. Conclusion and Future Work

We have presented Virtual Multiprocessor, a software layer that virtualizes a multi-processor machine on a commodity cluster. The experimental results show that our system achieved good performance for embarrassingly parallel coarse-grain tasks. Since this kind of parallel programs include various useful applications such as parameter sweep applications and parallel `make`, our system facilitates the wide deployment of commodity clusters.

As mentioned in Section 6, we plan a number of extensions and improvements to our system. Furthermore, we plan to evaluate our system using real-world applications such as SPLASH-2 and Apache.

The prototype implementation of Virtual Multiprocessor is available at `http://www.yl.is.s.u-tokyo.ac.jp/~kaneda/vmp`.

References

[1] Advanced Micro Devices. *AMD64 Virtualization Codenamed "Pacifica" Technology Secure Virtual Machine Architecture Reference Manual*, 2005.

[2] Y. Aridor, M. Factor, and A. Teperman. cJVM: a Single System Image of a JVM on a Cluster. In *Proc. of ICPP*, pages 4–11, 1999.

[3] A. Barak and O. La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Journal of FGCS*, 13(4-5):361–372, March 1998.

[4] 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 *Proc. of SOSP*, pages 164–177, 2003.

[5] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. of PPoPP*, pages 168–176, 1990.

[6] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proc. of SOSP*, pages 143–156, 1997.

[7] M. Chapman and G. Heiser. Implementing Transparent Shared Memory on Clusters Using Virtual Machines. In *Proc. of the USENIX Annual Technical Conference*, pages 383–386, 2005.

[8] Cooperative Linux. `http://www.colinux.org/`.

[9] E. N. (Mootaz) Elnozahy and Lorenzo Alvisi and Yi-Min Wang and David B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[10] H. Eiraku and Y. Shinjo. Running BSD Kernels as User Processes by Partial Emulation and Rewriting of Machine Instructions. In *Proc. of BSDCon*, pages 91–102, 2003.

[11] H.-J. H$\bar{\text{o}}$xer, K. Buchacker, and V. Sieh. Implementing a User-Mode Linux with Minimal Changes from Original Kernel. In *Proc. of Linux-Kongress 2002*, pages 72–82, 2002.

[12] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2003.

[13] Intel Corporation. *Intel Virtualization Technology Specification for the IA-32 Intel Architecture*, 2005.

[14] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[15] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the USENIX Winter Technical Conference*, pages 115–131. USENIX, 1994.

[16] S. T. King, G. W. Dunlap, and P. M. Chen. Operating System Support for Virtual Machines. In *Proc. of the USENIX Annual Technical Conference*, pages 71–84, 2003.

[17] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. of ICDCS '88*, pages 104–111, 1988.

[18] Microsoft Virtual PC. `http://www.microsoft.com/windows/virtualpc/`.

[19] C. Morin, R. Lottiaux, G. Vallée, P. Gallard, G. Utard, R. Badrinath, and L. Rilling. Kerrighed: a Single System Image Cluster Operating System for High Performance Computing. In *Proc. of Euro-Par*, pages 1291–1294, 2003.

[20] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proc. of the USENIX Security Symposium*, pages 129–144, 2000.

[21] O. Sato, R. Potter, M. Yamamoto, and M. Hagiya. UML Scrapbook and Realization of Snapshot Programming Environment. In *Proc. of ISSS*, pages 281–295, 2003.

[22] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proc. of ASPLOS-VII*, pages 174–184. ACM, October 1996.

[23] SCore Cluster System Software. `http://www.pccluster.org/`.

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

[25] The Message Passing Interface (MPI) standard. `http://www-unix.mcs.anl.gov/mpi/`.

[26] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Trans. on Computer Systems (TOCS)*, 14(1):80–107, 1996.

[27] User Mode Linux. `http://user-mode-linux.sourceforge.net/`.

[28] Virtual Iron Software. `http://www.virtualiron.com/`.

[29] VMware Inc. `http://www.vmware.com/`.

[30] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proc. of OSDI*, pages 181–194, 2002.

[31] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proc. of OSDI*, pages 77–90, 2004.

[32] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proc. of OSDI*, pages 195–209, 2002.