Mercury: Combining Performance with Dependability Using Self-virtualization *

Haibo Chen,Rong Chen,Fengzhe Zhang,Binyu Zang
Parallel Processing Institute
Fudan University
{hbchen,chenrong,fzzhang,byzang}@fudan.edu.cn

Pen-Chung Yew
Dpt. of Computer Science and Engineering
University of Minnesota at Twin-Cities
yew@cs.umn.edu

Abstract

There has recently been increasing interests in using system virtualization to improve the dependability of HPC cluster systems. However, it is not cost-free and may come with some performance degradation, uncertain QoS and loss of functionalities. Meanwhile, many virtualization-enabled features such as online maintenance and fault tolerance do not require virtualization being always on. This paper proposes a technique, called self-virtualization, that supports dynamically attaching and detaching a full-fledged virtual machine monitor (VMM) beneath an operating system, without disturbing applications thereon, and rid the system of potential overhead when the virtualization is not needed. This technique enables HPC clusters to reap most benefits from virtualization without sacrificing performance. This paper presents the design and implementation of Mercury, a working prototype based on Linux and Xen VMM. Our performance measurement shows that Mercury incurs very little overhead: about 0.2 ms *to complete a mode switch, and negligible performance degradation compared to Linux.*

1. Introduction

In the past few years, virtualization [10] has gained a resurgent popularity. Being aware of the benefits from system virtualization in improving system dependability, there has been a recent trend in applying it to high-performance computing (HPC) systems [17, 25, 5] and cluster computing [12, 13].

However, virtualization is not cost-free. Most general virtualization techniques usually bring some performance degradation [16, 15, 19] due to the additional layer of abstraction (i.e. VMM). For example, a recent measurement [21] indicates that SMP virtual machines have poor scalability that the performance for a 16-way SMP guest virtual machine is even not comparable to a uni-processor virtual machine. Such abstraction may also incur uncertain QoS (e.g. latency) for some highly concurrent services [18], conflict with assumptions for nowadays software and hardware architectures [8] (e.g. inconsistent disk storage in VMware [24]) and restrict functionalities due to restricted interface supported by VMMs.

Meanwhile, many virtualization-enabled features for dependability are only required occasionally. They include online hardware/software maintenance [14], checkpointing and restarting of operating systems [9, 22], live updating operating system kernels [6], among others. As an illustration, for online software maintenance, the VMM is only required during the maintenance process [14]. Placing a VMM beneath the operating system all the time will incur unnecessary performance degradation.

Being aware of unnecessary overhead of virtualization, Lowell et al. proposed Microvisor [14] for online software maintenance without an always-on VMM in a cluster environment. Facilitated by ALPHA architecture and redundant hardware, Microvisor supports at most two virtual machines and no memory and I/O virtualization. However, as a hardware-based VMM, Microvisor is tightly bounded to ALPHA and does not provide a general solution.

This paper proposes "self-virtualization", or "on-demand virtualization", a *general software-only* framework to avoid virtualization overhead during normal operations. This technique aims to eliminate the overhead induced by virtualization during normal execution, yet enjoys most of its benefits when needed. It enables an operating system to virtualize itself, as needed, through dynamically attaching a full-fledged virtual machine monitor(VMM) underneath, and detaching it when no longer needed. The added VMM can function as a normal full-fledge hypervisor that supports most general functions of a VMM. The virtualizing process is reversible so that the operating system can quickly switch its execution to run on a VMM and bare hardware.

We have built a working prototype, named Mercury, to provide self-virtualization capability to Linux running on Xen [4], a popular open-source VMM. To render our so-

*This work was funded by the 973 Plan under grant numbered 2005CB321905 and Intel University Research Grant.

lution more general and portable, Mercury is implemented by extending the virtual machine interface [23, 3], allowing Mercury independent of operating system evolutions to a great extent. According to our performance measurements, switching Linux between virtual mode (i.e. running on a VMM) and native mode (i.e. running on bare hardware) can be done in about *0.2 ms* without disturbing the running applications. Performance benchmarks show that Mercury in native mode incurs negligible performance overhead compared to native (i.e. unmodified) Linux.

The rest of this paper is organized as follows: in next section, we compare Mercury with existing systems; Section 3 describes the overall design of the framework. Section 4 discusses the implementation issues. Then we bring out the experiment results of Mercury in section 5. Finally, we close the paper with a brief conclusion.

2. Related Work

While there are many systems and innovations for system virtualization, our work mainly differs from the previous efforts in two aspects. First, Mercury is one of the first (if not the first) systems that allow an operating system to dynamically attach and detach a *full-fledged* VMM underneath. Second, the approach advocated by Mercury is purely software-based and requires no dedicated hardware support, which yields good portability and compatibility.

The most relevant work is Microvisor [14] developed at HP. Microvisor is a lightweight hardware-based VMM. It supports de-virtualizing and re-virtualizing the CPU state of the underlying Alpha 21264 microprocessor, and uses redundant hardware to support I/O partitioning with no support for memory virtualization. At most two VMs could be supported. It is dedicated to online software maintenance. This approach is tightly bound to the Alpha architecture and thus lacks both portability and scalability. Further, Microvisor is too lightweight to support more general virtualization techniques such as live migration [7], checkpoint/restart [9] and the ability to host multiple operating systems. In contrast, Mercury allows dynamically attaching and detaching a *robust*, *full-fledged* VMM, hence, it provides higher portability, scalability and robustness.

Virtual machine interface (VMI) [3] is a paravirtualization interface proposed by VMware, aiming to improving the portability and maintainability of existing virtualization solutions. Paravirt-ops [23] achieves OS portability by providing separate operation sets for Linux running on bare hardware and VMMs. However, their solutions allow no dynamically attaching and detaching a VMM underneath. Mercury is implemented in a similar way to VMI and Paravirt-ops, with additional support to in-flight attaching and detaching of a VMM underneath an running operating system.

Hardware vendors have also shipped hardware extension aiming to lower the virtualization overhead, such as Intel's Vanderpool [11] and AMD's Pacifica [2]. However, a recent study [1] indicates that existing hardware virtualization supports show little performance advantage over software-based virtualization system.

3. Self-virtualization of Operating Systems

This section presents a general framework of Mercury. We begin with the key difference between an operating system on bare hardware (native OS) and that on a VMM (virtualized OS), followed by key issues in providing an operating system with self-virtualization capabilities. Then, we present the general architecture of Mercury.

3.1. Mode Switch Between a Native OS and a Virtualized OS

Conventional operating systems lie in the lowest layer of the software stack, with direct control over hardware like CPU, memory and I/O devices. In contrast, in a virtualized environment, the VMM manages all hardware resources and exposes them to operating systems thereon in the form of virtual machines [1]. Hence, some portions of operating system code behaves differently between a native OS and a virtualized OS. We clarify the key differences to facilitate the implementation of mode switches of the operating systems. Here, a mode switch refers to a transition of the operating system execution between native mode and virtual mode, which requires an adjustment of OS code and data to suite the corresponding execution mode.

3.1.1 CPU Privilege Level

Modern computers usually provide some protection mechanisms to prevent arbitrary accesses to hardware state. Most hardware state is accessible only in the most privileged level via privileged instructions. General virtualization techniques usually involve de-privileging operating systems [4, 20]: making VMMs executing at the most privileged level and leaving operating systems to execute at less privileged levels.

Hence, operating systems running in virtual mode and native mode differ in their privilege levels and their means to access the hardware resources. Therefore, some portions of operating system code behaves differently in different execution modes. This is especially true for *virtualization sensitive code and data*. *Virtualization sensitive data* stores the hardware control state and operating systems control

[1] Here, we only focus on VMMs which directly execute on bare hardware. VMMs running on a host operating system (e.g. VMWare workstation) are beyond the scope of this paper.

state that varies in different execution modes, such as CPU control state and page tables. *Virtualization sensitive code* refers to the code that manipulates such data structures, examples include sensitive instructions and operations on sensitive memory(e.g. page table updates). When running on bare hardware, operating systems directly execute the virtualization sensitive code; while operating systems execute on VMMs, they have to rely on the services provided by the VMMs.

3.1.2 Address Space Layout

For a self-virtualization system, an operating system in virtual mode differs from its native counterpart in both virtual address space and physical address space. Generally, OS kernel and a user process reside at the same virtual address space. In a virtualized system, a VMM coexists with the OS kernel and user processes. As in a computer (such as x86) with hardware-managed TLB, flushing TLB due to address space switches is rather costly, modern virtualization techniques usually place the VMM, OS kernel and a user process in a single address space. For example, Xen VMM occupies the top 64-MB virtual address in a single 4-GB virtual address space. Therefore, for a virtualized OS, the kernel address space layout is different from a native OS. As a dynamic adjustment of the address space layout is rather time consuming, Mercury instead unifies the address space layout to achieve a smooth transfer between native mode and virtual mode, by reserving a fixed portion of virtual address space for the VMM.

For physical address space, most commodity operating systems assume the continuity of the whole physical memory. However, in a virtualized environment, as there are multiple operating systems, their physical memory are discontinuous. To ensure correct system behavior, two physical address modes are available in modern virtualization systems: shadow mode and direct mode. In shadow mode, a VMM presents the guest operating systems an illusion of contiguous pseudo-physical memory and is responsible for translating pseudo-physical memory to physical memory. Thus, a translation from pseudo-physical memory to physical memory is required during a self-virtualization. In direct mode, a VMM provide direct accesses to page tables for guest operating systems: page tables in guest operating systems are directly installed in MMU but only read accesses are granted. As the page table entries in guest operating systems are directly installed in hardware, no translation is required during a mode switch, which could largely reduce the complexity of implementing a self-virtualization system. Currently, Mercury utilizes the direct access mode to simplify the implementation.

3.1.3 Memory Management

In a virtualized environment, a VMM should track the usage of all pages to ensure strict isolation among virtual machines. Consequently, when transferring an OS between native mode and virtual mode, Mercury should ensure the consistency of VMM's memory management information. In addition, the access mode to the MMU differs between a native OS and a virtualized OS. A native OS can directly access all MMU, while a virtualized OS should rely on the services of a VMM. For example, updates to page tables could be directly done in native mode, while in virtual mode, they need to either invoke the interface provided by VMMs or rely on a trap-emulation service in VMMs.

3.1.4 I/O Access Modes

A fully virtualized OS usually has no direct control over I/O devices. A VMM is in charge of managing the I/O devices and exposing them to operating systems either via implicit trap and emulation [20] or via explicit services [4]. As I/O emulation tends to be time-consuming, for the sake of performance, device drivers in a para-virtualized OS are usually modified to explicitly invoke the interface provided by the VMM. For example, Xen provide a splitted I/O mode [4] in a frontend/backend manner for device accesses in a virtual machine.

3.2 Key Issues in Self-virtualization of Operating Systems

Being aware of the differences between a native OS and a virtualized OS, we identify several key aspects in self-virtualizing an operating system as follows.

3.2.1 Pre-caching of VMMs

The major challenge in the design and implementation of Mercury is to dynamically attach and detach a VMM beneath a running operating system, without disruption to the running applications. Mercury accomplishes this through a simple and common hardware mechanism: *interrupt*. The interrupt handler dedicated to self-virtualization contains routines to attach and detach a VMM on-the-fly. Execution mode switches can be done through triggering the corresponding interrupt line.

However, handling such interrupts should not be time-consuming; otherwise, other interrupts might be delayed or missed. Therefore, it is crucial that the interrupt handlers be very efficient. To satisfy such a requirement, it is unrealistic to boot a complete VMM on the fly. Instead, this process is optimized by warming up the VMM during the machine boot, and adding only a minimal amount of work to provide necessary state for hardware when the VMM is attached. As

a VMM occupies only a reasonably small chunk of memory, we believe it is worthy for such space-time tradeoff because it shortens the mode switch time from several seconds to several sub-milliseconds.

The Pre-cached VMM already contains most required data structures in memory. The only data structures required to be adjusted are those maintaining the state of the virtual machines thereon, including the in-time execution context, memory page type and count information, and interrupt bindings. All these data structures will be synchronized by Mercury during a mode switch using state reloading functions described in section 4.1.3.

3.2.2 Transparent Self-virtualization

As the virtualization sensitive code and data differ between a native OS and a virtualized OS, a relocation of such code and data is required during a mode switch. Further, to ensure the safety of self-virtualization, it is crucial to track whether it is safe to perform a mode switch or not, so that the kernel will not enter an undefined state in which some portions of the code execute in the native mode and others execute in the virtualized modes. Dynamic rewriting and para-virtualization technologies are two possible methods.

Dynamic rewriting technique [1] replaces sensitive instructions in operating system kernels at execution time. It could result in good OS transparency. However, deciding whether it is safe to perform such rewriting is extremely difficult. Further, binary rewriting all related code is rather time-consuming for a mode switch.

In contrast, para-virtualization statically modifies OS kernel to cooperate with VMM by replacing sensitive instructions with function calls to VMM. This approach will result in short switch time, and it is easy to track whether it is safe to perform a mode switch (e.g. by reference counting entrance/exit from virtualization sensitive code). However, such an approach will result in significant maintenance cost during the operating system evolution.

Mercury chooses the para-virtualization approach but in a portable and OS-transparent way similar to paravirt-ops [23] and VMI [3]. Specifically, Mercury groups all virtualization sensitive code and data, and defines a unified interface: a *virtualization object* composed of a function table and a data table. Mercury provides separate object implementation for operating systems executing in virtual mode and native mode. Relocation of virtualization sensitive code and data is done by changing the object pointer maintained by the operating system.

3.2.3 Maintaining Behavior Consistency

To completely shadow running applications from these mode transitions, the operating system should exhibit a consistent behavior regardless of its current mode. Thus, there are three requirements in ensuring behavior consistency: first, for virtualization sensitive code, it is required to ensure that its execution is conformed to current mode, e.g., code for native mode should not execute in virtual mode; second, for virtualization sensitive data, their state in each execution mode should be semantically equivalent, which requires they provide equivalent services to OS kernel and applications; third, for hardware control state, such as control registers, page tables, description tables, it should be reloaded accordingly during a mode switch. Details on how Mercury ensures these requirements are presented in section 4.1.

3.3 Architecture of Mercury

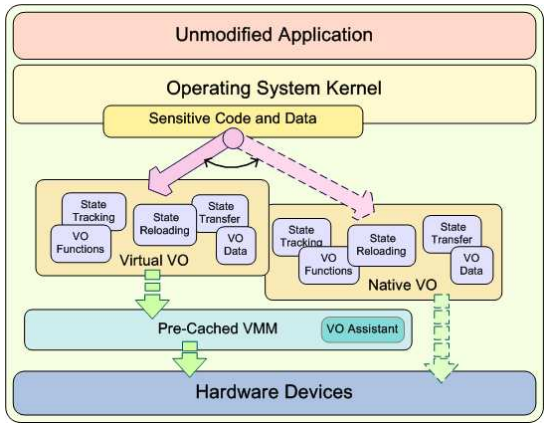

Figure 1. The architecture of Mercury.

Figure 1 depicts the architecture of Mercury. The key to self-virtualization is a variety of virtualization objects (VOes), which encapsulate virtualization sensitive code and data. The VOes are neutral to operating system upgrades but sensitive to VMM evolutions. The modularity of VOes enhances the maintainability of Mercury as it allows easy adaptation of Mercury to new VMMs and architectures.

To support a fast switch of operating systems from native mode to virtual mode, Mercury warms up a VMM during system initialization and always keeps it in memory. As generally a VMM is relatively small, the pre-cached VMM actually creates very little memory pressure. A VO instance in virtual mode relies on the services from pre-cached VMM while a VO instance in native mode directly manipulates the hardware. When an operating system is relocated from native mode to virtual mode, the pre-cached VMM is activated and takes over the hardware. The VO-assistant is composed of some help routines in the VMM. It provides services such as self-virtualization interrupt handlers to assist the VO instances to maintain the state during a mode transition.

4. Detailed Design and Implementation

We initially implemented Mercury based on Linux 2.6.10 running on Xen-2.0.5. Later, we port Mercury to newly Xen-3.0.2. The hardware platform is x86 architecture. We chose Xen as a base platform because of its open-source nature and robustness. Although our current implementation was specific to Linux on Xen for x86, we believe the architecture and the design of Mercury could be similarly implemented on other operating systems, VMMs and processor architectures.

The following subsections discuss some specific design and implementation of Mercury. First, we present in detail how to maintain a consistent state in a mode switch. Then, we provide some specific design on dynamical virtualization of I/O devices. Finally, we describe the implementation of *virtualization objects*.

4.1. Maintaining Behavior Consistency

4.1.1 State Tracking of Virtualization Sensitive Code

Mercury tracks the execution of virtualization sensitive code by reference counting the execution of a *virtualization object* on its entry and exit.

Mercury applies a mode switch only when the reference counter reaches zero. One potential problem is that mode switch requests may sometimes fail if some counters is non-zero at that time. However, due to the fact that almost all execution in the *virtualization object* is short (because it is non-blocking) or synchronous, this problem rarely happens. If such a condition does occur, Mercury registers a timer to the OS kernel. The timer checks if the reference counter reaches zero in every time interval (e.g. every 10 ms). If so, the mode switch will be safely committed.

4.1.2 State Transfer of Virtualization Sensitive Data Structures

Mercury utilizes *state transfer* functions to efficiently transfer the state of virtualization sensitive data from one mode to the other during a mode switch, to ensure that they provide equivalent services.

There are several key sets of state in OS kernel that must be transferred during a mode switch: (1) page table pages, which are read-only in the virtualized modes while writable in the native mode; (2) the privileged level of the kernel segment in each kernel thread, whose value is 0 in native mode and 1 in virtualized modes; (3) the interrupt handlers and interrupt bindings (e.g. APIC, I/O APIC), which directly manipulate the hardware in the native mode while rely on the service of VMMs in the virtual modes. Mercury provides a set of state transfer functions, which are responsible for transferring the state of the virtualization sensitive data structures during a mode switch.

Maintaining behavior consistency for VMM's memory management poses additional challenges. To ensure secure isolation among guest operating systems, Xen provides a rather complex page management interface and maintains the owner, type and count information for each page frames. When a VMM is active, it has to track the usage of all page frames to ensure correct usage of each page. In native mode, as the VMM is inactive it will lose track on the usage information of these pages. Thus, it is required to correctly refill these information for the VMM to enforce correct system behavior. Generally, there are two alternatives to ensure the consistency of the underlying VMM: one is to actively adapt the count information in a VMM each time an OS modifies its page tables; the other is to re-compute and synchronize the information during a mode switch. The first approach incurs some performance overhead in native mode, while shortens some time during a mode switch.

We have implemented both approaches for the memory management of Xen. According to our performance experiment, the first approach will incur about 2%-3% performance overhead and saves only a small amount of mode switch time. Hence, we preferably choose the latter approach.

It should be noticed that some state cached in stack is not easy to provide proper state transfer functions. In practice, an interrupted thread will push their interrupt context in the thread stack. The code and data segment selectors pushed in the thread stack contain the privilege level information of the operating system. If a mode switch occurs here, the resumed thread will pop the saved segment selectors and trigger a general protection fault. This problem is solved by adding a code stub to check and fix the cached segment selectors.

4.1.3 State Reloading of Hardware Control State

The state of the underlying hardware usually differs in different execution modes. Therefore, when switching from a virtual mode back to the native mode, the control state should be reloaded into the hardware accordingly. General state includes the base pointer of a page table, interrupt tables, descriptor tables (e.g. GDT, LDT), among others.

Since the critical state of the hardware is modified in the state reloading process, the reloading process must not be interrupted. Hence, Mercury adds two interrupt handlers for mode switches between the native mode and the virtual mode, and applies the reloading in the interrupt context. The interrupt handlers will reload the state, invoke the state transfer functions to transfer the state of operating systems, and return to operating systems.

However, the interrupt handlers are slightly different

from general interrupt service routines which return to the previous privileged level after service completion. In Mercury, VMMs and a native OS lie in the most privileged level (e.g. PL0), while a virtualized OS executes in the next privileged level (e.g. PL1). Therefore, there is a privileged-level switch right after a mode switch. This is accomplished by modifying the privileged level in the return stack of the interrupt.

4.2. Self-virtualizing an SMP OS

Self-virtualizing a SMP-OS poses additional challenges compared to a uni-processor OS. Processors should be coordinated to avoid that they execute in different modes. Mercury uses IPI (inter-processor interrupt) mechanism and shared variables to control the mode switch of each processor.

The processor (CP, control processor) received the mode switch request will notify other processors via issuing IPIs. Upon receiving the IPI, each processor notifies its readiness to other processors by increasing a shared count and waits for a shared flag to ensure all other processors are ready to do mode switch. The shared flag will be set by the CP when it finds the shared count is equal to the total number of processors. The completion of the mode switch is also coordinated using a shared variable.

4.3. Device Virtualization

In Xen VMM, only driver domain (usually domain0) has direct accesses to the hardware devices, while other production domains (domainU) access the hardware through the interface provided by domain0 in a frontend/backend fashion: the frontend drivers in domainU serve the hardware access requests by forwarding the requests to the backend drivers in domain0 using shared-memory I/O rings; the backend drivers invoke services from the hardware to serve the requests and forward the results back to the frontend drivers in domainU; when the domainU is migrated, the frontend drivers reconnect themselves to the new backend drivers on the new host machine. Thus, the decoupling of frontend/backend drivers provides the mobility to the device drivers.

To host multiple VMs in the self-virtualized OS, the OS serves as the driver domain and hosts the backend drivers. For live migration of VMs, since current live migration systems often rely on networked file system, disk drivers are essentially migratable. For network devices, since the packets loss during the migration could be solved at the network protocol level, Mercury currently doesn't decouple the network device drivers *before* the migration. Instead, it creates the frontend device drivers and connects them to the backend drivers *after* the migration has been completed.

4.4. Virtualization Object

Each VO instance is composed of the corresponding implementation of virtualization sensitive code and data for an execution mode, as well as some additional components to support self-virtualization of an operating system. Such components are responsible for relocating the execution of operating systems in different execution modes and maintaining the behavior consistency after a dynamic relocation of virtualization instances. We have implemented a native VO and a virtual VO for Linux and Xen VMM accordingly. Each such VO is a structure with a set of function tables and corresponding data in essence. The data in a VO includes some global sensitive data, such as a set of control registers, descriptor tables (IDT, GDT and LDT). The function tables are composed of functions for virtualization sensitive code and those for self-virtualization.

Functions for virtualization sensitive code are abstractions of sensitive operations: sensitive CPU operations, which manipulate the privileged state of CPU, such as privilege levels and interrupt flags; sensitive memory operations, which modify page tables; sensitive I/O operations, which access the device resources through memory-mapped I/O or I/O port. A function in a native VO directly manipulates the hardware while it invokes interfaces provided by the VMM (e.g hypercalls in Xen VMM) in virtual mode. To maintain state consistence, all of these functions are reference-counted to track the execution of operating systems in a VO. Note that non-performance-critical sensitive code are not included in a VO and relies instead on trap-and-emulation to commit the effect.

Functions for self-virtualization consist of state-transfer functions to transfer the state of virtualization sensitive data structures during a mode switch, and state reloading functions to relocate the execution mode of an operating system and activate/deactivate the pre-cached VMM, as described in section 4.1.3 and section 4.1.2.

5. Evaluation

In this section, we present the performance results of Mercury. As our implementation is based on Xen VMM, we test Mercury against Xen-Linux and native Linux running on bare hardware to assess overall performance of Mercury. We compare the performance of Mercury-Linux (Linux running on Mercury) in native mode and virtual mode (M-N and M-V accordingly) against native Linux (N-L) and Xen-Linux (both control domain, domain0 (X-0) and production domain, domainU (X-U)). Since Mercury allows a self-virtualized operating system to host unmodified Xen-Linux (M-U), we present its performance results as well. Further, the timer frequency is 100Hz for all systems.

As it is crucial that switch time among different execution modes be minimal, we present the measured switch time as well.

5.1. Experimental Setup

All experiments were conducted on a system equipped with a 3.0GHz Pentium IV with 1GB SDRAM, a Realtek r8169 1 Gigabit Ethernet NIC , and a single 250GB 7200 RPM SATA disk, with 20GB allocated to each Linux distribution. The version of Linux, Xen-Linux and Mercury-Linux is 2.6.16 and the version of Xen VMM is 3.0.2. The Fedora Core 2 distribution was used throughout. It is installed on ext3 file system. 900,000KB of memory is given to each variant of Linux except unprivileged domain (i.e. domainU). DomainU is configured to 870,000KB of memory as it relies on Domain0 to complete a device access. Therefore, we decrease the memory reservation in domainU to even this unfairness. The Linux running as the production domain in both Xen and Mercury is configured to use a 20GB partition in the same disk with the ext3 file system as well. The disk is used in "raw mode", which is believed to have the best performance.

For application level benchmarks, we present the performance results for the Open Source Database Benchmark suite(OSDB), dbench , Linux build time. OSDB evaluates the performance of PostgreSQL database, with the test for Information Retrieval (IR). For the experimental setup, we used OSDB-x0.15-1 in conjunction with PostgreSQL 7.3.6. Dbench is a strict I/O bound benchmark, the version used is 3.03. Linux build time measures the overall time to built a Linux Kernel 2.6.16 with gcc-3.3.3. For micro-benchmarks, we measured the *lmbench* benchmark of version 3.0-a5, and report the results for the OS-related parts for all seven systems. We also tested Unixbench 4.1.0 to gain the overall system performance results. For network performance, we used iPerf to measure the bandwidth with TCP and UDP traffic, the client and server for iPerf were connected through a Giga-bit switch. All benchmarks were with their default configurations. In addition, we investigated the time spent to apply each mode switch. All benchmarks were tested 5 times and each result is an average of them.

5.2. Overall Performance

Figure 2 depicts the overall performance of Mercury against native Linux and Xen-Linux. Table 1 shows the OS-related results of lmbench. Our measurements indicates that although a variety of optimizations have been integrated into Xen, there is still some performance degradation compared to native Linux, especially for *memory* and *I/O intensive* applications. For example, *mmap* in lmbench incurs

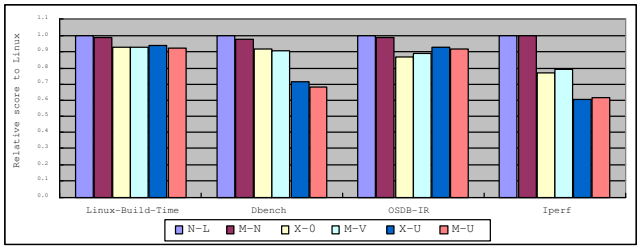

Figure 2. Relative Performance of Mercury Against Linux and Xen-Linux

57% performance loss while for process creation the result is 72%. For application level benchmarks, domain0 (X-0) shows 10% performance degradation for dbench, while domainU (X-U) incurs 30% performance loss. Both domain0 and domainU incur about 7% for Linux kernel build, and 20% for OSDB-IR. As the Xen architecture has evolved dramatically in Xen2 and Xen3, the results are somewhat biased with the early results of Xen [4]. However, our results are mostly conformed to a recent measurement by Soltesz et al. [19]. In contrast, the performance of Mercury in its two modes is nearly the same compared to native Linux, domain0 and domainU accordingly. The source of performance loss in Mercury mainly lies in the changes to code and data layout and function calls to *virtualization objects*. However, as shown in the figure, such overhead is negligible.

5.3. Mode Switch Time

We measured the time to apply a mode switch by reading the hardware cycle counter register (using the RDTSC instruction) at both the beginning and the end of each mode switch. Table 2 shows the time to apply a mode switch between native mode (N) and virtual mode in nanosecond. As depicted in the table, the time spent in a mode switch is relatively small: the average time is about *0.2 ms* at most.

From the table, it can be seen that the time spent to switch from native mode to virtual mode is much longer than the time to switch back. This is expected, as mentioned in section 4.1.2, Mercury has to recalculate the type and count information for all page frames during a mode switch, which accounts for the major time to commit a switch. Nevertheless, the overall time is still relatively small and we believe it is acceptable as we can gain good performance during normal operations in native mode.

6 Conclusion

We have proposed a technique, called self-virtualization, that enables an operating system to dynamically attach and

Config	N-L	M-N	X-0	M-V	X-U	M-U
fork process	93	113	323	323	311	311
exec process	340	392	858	846	830	831
sh	1089	1220	2174	2158	2080	2082
ctx(2p/0k)	1.59	2.49	3.63	4.04	3.53	3.51
ctx(16p/16k)	2.29	3.56	4.38	5.30	4.24	4.36
ctx(16p/64k)	3.52	6.90	7.14	8.78	6.71	6.94
mmap	3380	3644	7778	7439	8406	8152
prot fault	0.55	0.61	0.92	1.01	0.97	0.98
page fault	1.19	1.48	2.64	2.66	2.59	2.57

Table 1. lmbench latency results - time in μs.

Mode	Native to Virtual	Virtual to Native
#1	199,800	59,517
#2	202,545	60,398
#3	206,192	62,630
#4	209,814	56,302
#5	233,268	59,943
avg.	210,234	59,758

Table 2. Time to apply a mode switch – in nanosecond.

detach a full-fledged VMM underneath. This approach is completely software-based and mostly OS-transparent. It effectively eliminates unnecessary performance overhead of system virtualization and thus combines performance and dependability in applying system virtualization to HPC clusters. Performance measurements show that such an implementation incurs negligible performance overhead and allows fast switches among different execution modes.

References

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proc. of ASPLOS-XII*, pages 2–13, 2006.

[2] Advanced Micro Devices. Secure virtual machine architecture reference manual. www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/33047.pdf, May 2005.

[3] Z. Amsden, D. Arai, D. Hecht, and P. Subrahmanyam. Virtual machine interface (vmi). http://www.vmware.com/interfaces/, March 2006.

[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'03*, pages 164–177. ACM, 2003.

[5] H. Bjerke. HPC Virtualization with Xen on Itanium. Master's thesis, NTNU, July 2005.

[6] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew. Live updating operating systems using virtualization. In *Proc. of VEE'06*, pages 35–44, Ottawa, Canada, June 2006.

[7] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. of NSDI*, pages 273–286, 2005.

[8] T. Garfinkel and M. Rosenblum. When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments. In *HOTOS-X*, 2005.

[9] H. O. Geoffroy Vallee, Thomas Naughton and S. L. Scott. Checkpoint/restart of virtual machines based on xen. In *HAPCW 2006*, Santa Fe, NM, USA, October 2006.

[10] R. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, 1974.

[11] Intel Cooperation. Intel vanderpool technology for IA-32 processors (VT-x) preliminary specification. http://www.intel.com/technology/computing/vptech/.

[12] N. Kiyanclar. A Survey of Virtualization Techniques Focusing on Secure On-Demand Cluster Computing. *Arxiv preprint cs.OS/0511010*, 2005.

[13] N. Kiyanclar, G. Koenig, and W. Yurcik. Maestro-VC: A Paravirtualized Execution Environment for Secure On-Demand Cluster Computing. In *Proc. of CCGRID'06*, 2006.

[14] D. Lowell, Y. Saito, and E. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proc. of ASPLOS-XI*, pages 211–223, 2004.

[15] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in xen. In *Proc. of Usenix'06*, pages 15–28, 2006.

[16] A. Menon, J. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proc. of VEE'05*, pages 13–23, 2005.

[17] M. Mergen, V. Uhlig, O. Krieger, and J. Xenidis. Virtualization for high-performance computing. *ACM SIGOPS Operating Systems Review*, 40(2):8–11, 2006.

[18] P. Padala, X. Zhu, Z. Wang, S. Singhal, and K. G. Shin. Performance evaluation of virtualization technologies for server consolidation. Technical Report HPL-2007-59, HP, 2007.

[19] S. Soltesz, H. Potzl, M. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *Proc. of EuroSys*, 2007.

[20] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proc. of USENIX'01*, pages 1–14, 2001.

[21] A. Theurer, K. Rister, O. Krieger, R. Harper, and S. Dobbelstein. Virtual Scalability: Charting the Performance of Linux in a Virtual World. In *Proc. of Linux Symposium*, 2006.

[22] VMware. The VMWare software package. *See http://www.vmware.com*, 2006.

[23] C. Wright. Para-virtualization interfaces. http://lwn.net/Articles/194340/, G 2006.

[24] J. Yang, C. Sar, and D. Engler. Explode: A lightweight, general system for finding serious storage system errors. In *Proc. of OSDI'06*, pages 131–146, 2006.

[25] L. Youseff, R. Wolski, B. Gorda, and C. Krintz. Paravirtualization for HPC Systems. Technical Report 2006-10, CSE, UCSB, Aug. 2006.