Chen HB, Zhang FZ, Chen R *et al.* Mercury: Combining performance with dependability using self-virtualization. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 27(1): 92–104 Jan. 2012. DOI 10.1007/s11390-012-1208-1

Mercury: Combining Performance with Dependability Using Self-Virtualization

Hai-Bo Chen[1] (陈海波), *Member*, *CCF*, *ACM*, *IEEE*, Feng-Zhe Zhang[2] (张逢喆), *Member*, *CCF*, *ACM*
Rong Chen[2] (陈　榕), Bin-Yu Zang[2,*] (臧斌宇), *Senior Member*, *CCF*, *Member*, *ACM*
and Pen-Chung Yew[3] (游本中), *Fellow*, *IEEE*

[1] *School of Software, Shanghai Jiaotong University, Shanghai 200240, China*

[2] *Parallel Processing Institute, Fudan University, Shanghai 201203, China*

[3] *Institute of Information Science, Academia Sinica, Taipei, China*

E-mail: haibochen@sjtu.edu.cn; {fzzhang, chenrong, byzang}@fudan.edu.cn; yew@iis.sinica.edu.tw

Received December 15, 2010; revised October 7, 2011.

Abstract Virtualization has recently gained popularity largely due to its promise in increasing utilization, improving availability and enhancing security. Very often, the role of computer systems needs to change as the business environment changes. Initially, the system may only need to host one operating system and seek full execution speed. Later, it may be required to add other functionalities such as allowing easy software/hardware maintenance, surviving system failures and hosting multiple operating systems. Virtualization allows these functionalities to be supported easily and effectively. However, virtualization techniques generally incur non-negligible performance penalty. Fortunately, many virtualization-enabled features such as online software/hardware maintenance and fault tolerance do not require virtualization standby all the time. Based on this observation, this paper proposes a technique, called Self-virtualization, which provides the operating system with the capability to turn on and off virtualization on demand, without disturbing running applications. This technique enables computer systems 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 virtual machine monitor. The performance measurement shows that Mercury incurs very little overhead: about 0.2 ms on 3 GHz Xeon CPU to complete a mode switch, and negligible performance degradation compared to Linux.

Keywords dependability, performance, self-virtualization, dynamic virtualization

1 Introduction

System virtualization[1] has been widely adopted to assist resource management[2-6], enhance system security[7-10], hardware/software maintenance[11-12], HPC systems[13-15], energy saving[16-19], as well as recent cloud computing[20-21]. Hardware vendors also shipped new features to assist virtualization[22-24].

Despite numerous hardware and software efforts in improving the performance of virtualization, virtualization is still not cost-free. General virtualization techniques usually incur non-negligible performance overhead, which may be intolerable for some performance critical applications such as high performance computing[25]. For example, several previous measurements show that virtualization still incurs high overhead and degraded quality of services (QoS) for many applications, especially for multicore and high-performance applications[26-28]. For example, a performance measurement from Walker *et al.*[29] shows that, Xen-based[30] Amazon EC2 platform[20] incurs around 40%∼1 000% performance overhead for NASA applications. A performance study against the recent Xen-4.0.0 on multicore machines also shows that Xen incurs up to 20X overhead for some multicore applications due to contentions inside the Xen VMM (virtual machine monitor) and increased iTLB and cache misses[28].

Further, with the scale of inter-connected computers and the number of cores in a computer continuously increasing, hardware failure rates also increase exponentially[31-32]. Online maintenance, system

Regular Paper

This work was supported by the National Nature Science Foundation of China under Grant Nos. 61003002 and 90818015, the Science and Technology Commission of Shanghai under Grant No. 10511500100, Fundamental Research Funds for the Central Universities in China, and Shanghai Leading Academic Discipline Project under Grant No. B114.

[*] Corresponding Author

mobility and other system virtualization enabled features are desirable. Further, the promising convenience of system management brought by virtualization is desirable[33]. However, the performance degradation incurred by system virtualization is intolerable in the territory where performance is critical, including HPC clusters, scientific Grid and PlanetLab[34].

Meanwhile, many virtualization-enabled features for dependability are only required occasionally. They include online hardware/software maintenance[11], checkpointing and restarting of operating systems[35-36], live updating operating system kernels[12], among others. As an illustration, for online software maintenance, the VMM is only required during the maintenance process[11]. Placing a VMM beneath the operating system all the time will incur unnecessary performance degradation.

Being aware of the unnecessary overhead of virtualization, Lowell *et al.* proposed Microvisor[11] 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 at eliminating 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 between on a VMM and on bare hardware.

We have built a working prototype, named Mercury, to provide self-virtualization capability to Linux running on Xen[30], a popular open-source VMM. To render our solution more general and portable, Mercury is implemented by extending the virtual machine interface[37-38], allowing Mercury to be 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 on 3 GHz Xeon CPU 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 gives an overview of system virtualization and describes the key difference in the behavior of an operating system on a bare metal and a virtualized platform, which motivate our overall design of Mercury in Section 4. Section 5 describes the implementation issues, followed by a discussion on possible usage scenarios of self-virtualization in Section 6. Then we bring out the experimental results of Mercury in Section 7. Finally, we conclude the paper with a discussion of possible future work.

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 self-virtualization technique is purely software-based and requires no dedicated hardware support, thus yields good portability and compatibility.

The most relevant work is Microvisor[11] 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[39], checkpoint and restart[35] and the ability to host multiple operating systems. In contrast, Mercury allows dynamically attaching and detaching a *full-fledged* VMM, hence, it provides higher portability and scalability.

Contemporary virtualization systems are dominated by two trends: full system virtualization and para-virtualization. VMware[36] and Microsoft Hyper-V[40] are well known full virtualization systems. Unlike full system virtualization that provides a fully abstract virtual machine interface, para-virtualization exposes a part of hardware to hosted operating systems. Such exposure compromises transparency of operating systems to reduce performance penalty of virtualization. Denali[41] and Xen[30] are typical para-virtualization systems. KVM[42] is a recent Linux kernel module based virtualization systems. It is a hosted VMM that provides both para-virtualization and full system virtualization support (require Intel VT[43] or AMD-V[44]). Different from their design goals of optimizing a VMM for workload consolidation, Mercury intends to eliminate the virtualization overhead when a VMM is not required.

Para-virtualization incurs additional maintenance efforts with the evolving of virtual machine interface (VMI). [38] is a para-virtualization interface proposed by VMware, aiming at improving the portability and maintainability of existing virtualization solutions. Paravirt-ops[37] achieves OS portability by providing separate operation sets for Linux running on bare hardware and VMMs. However, their solutions do not allow 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 a running operating system.

The popularity of virtualization has boosted the commercial hardware enhancements that reduce the complexity of virtualization and improves performance. They include CPU virtualization such as Intel's Vanderpool[43] and AMD's Pacifica[44], memory virtualization nested or extended page table (EPT[22] or NPT[24]), and I/O virtualization[45]. However, using hardware-assisted virtualization might also tightly bind one virtual machine on a platform, limiting its mobility across multiple platforms, with different virtualization architectures and features.

Alternatives are proposed to address the overhead of virtualization. Container-based operating system virtualization[34] strives to present HPC clusters, the Grid, hosting centers, and PlanetLab with both isolation and efficiency. As indicated in the name, the application level container does not support operating system level virtualization features like VM migration.

3 Background on Virtualization

This section first describes the general architecture of system virtualization, then identifies the key differences for an operating system running on a VMM and a bare metal.

3.1 General Virtualization Architecture

As depicted in Fig.1, a VMM is a software layer lying between operating systems and hardware[①]. It manages the underlying hardware resources and exposes the resources to the above operating systems in the form of a virtual machine (VM)[1]. For the sake of performance, a VMM only intercepts privileged instructions that change or access the hardware state, leaving other instructions running without intervention. To provide strong isolation among various VMs, the interception of privileged instructions is mandatory and cannot be bypassed. On commodity processors, the interrupt lines and some process control states (e.g., page table root) are usually privileged states, thus accesses to them should be intervened by the VMM.

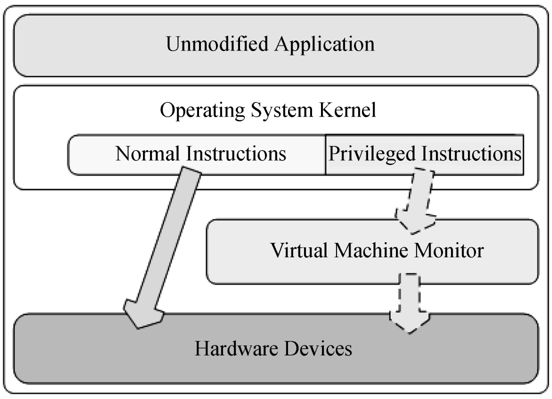


Fig.1. General structure of virtualization.

3.2 Differences Between a Native OS and a Virtual 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. Hence, some portions of operating system code behave 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.

3.2.1 *CPU Privilege Level*

Modern computers usually provide some protection mechanisms to prevent arbitrary accesses to hardware state. Most hardware state is accessible only at the most privileged level via privileged instructions. General virtualization techniques usually involve de-privileging operating systems[30,46]: making VMMs executing at the most privileged level and leaving

[①] In this paper, we only focus on VMMs which directly execute on bare hardware.

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 behave 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 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.

3.2.2 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 is 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 provides direct accesses to page tables for guest operating systems: page tables in guest operating systems are directly installed in memory management unit (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.

3.2.3 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.

3.2.4 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[46] or explicit services[30]. 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 provides a splitted I/O model[30] in a frontend/backend manner for device accesses in a virtual machine.

4 Self-Virtualization: Approaches and the Architecture

Being aware of the differences between a native OS and a virtualized OS, we first identify several key aspects in self-virtualizing an operating system as well as our solutions. Then, we present the architecture of our system.

4.1 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 Subsection 5.1.3.

4.2 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 portion of the code execute in the native mode and others execute in the virtualized mode. Dynamic rewriting and para-virtualization technologies are two possible methods.

Dynamic rewriting technique[47] 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[37] and VMI[38]. 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.

4.3 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 them to 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 Subsection 5.1.

4.4 Architecture of Mercury

Fig.2 depicts the architecture of Mercury. The key to self-virtualization is a variety of virtualization objects

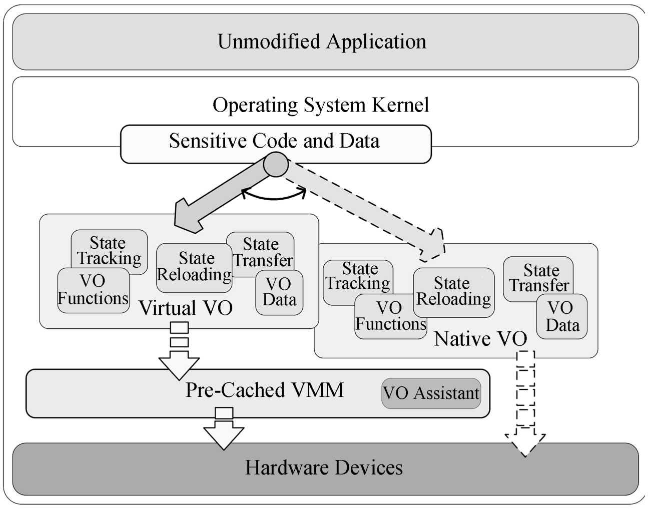


Fig.2. Overall architecture of Mercury.

(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 control of 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.

5 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 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*.

5.1 Maintaining Behavior Consistency

5.1.1 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 are 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.

5.1.2 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 frame. 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 this 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 it 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.

5.1.3 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 at the most privileged level (e.g., PL0), while a virtualized OS executes at 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.

5.2 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 does not 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.

5.3 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 is 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 Subsections 5.1.2 and 5.1.3.

5.4 Supporting Multicore Machines

Self-virtualizing a multicore OS poses additional challenges compared to a uni-processor OS. Multiple

CPU cores should be coordinated to avoid state inconsistency where cores 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) that 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 a 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.

6 Usage Scenarios

Self-virtualization could mitigate the performance overhead caused by virtualization while retain its full benefit. In this section, we discuss its possible applications to increase system dependability, yet with little performance degradation: online hardware maintenance, live updating operating system kernel code and data, and improving the availability of clusters. We believe these enhancements are rather important for current clusters and some cloud computing systems.

6.1 Checkpointing and Restarting of Operating Systems

Checkpointing and restarting of computing environments are widely deployed to increase system dependability. By checkpointing the execution environment periodically and restarting the execution from a specific checkpoint during a failure, they provide proactive fault-tolerant features to many mission-critical systems.

While virtualization could provide checkpoint/restart at operating system level[35-36], it also brings some performance overhead. We argue that self-virtualization could eliminate such overhead. To perform checkpointing, the pre-cached VMM is activated and makes a snapshot of the whole system, then the VMM is detached and remains inactive. If a software failure occurs, the VMM could be automatically reactivated to restore the failed system into a recent checkpoint. For hardware failures, the snapshot could be manually restored to another healthy machine.

6.2 Self-Healing of Operating Systems

Self-healing of a computing system has gained prevalence in system research recently[48-49]. Being aware of the limitation of "healing-from-within" approach, Bohra *et al.*[49] use a remote computer to repair the software state over Myrinet network. However, as the approach requires an additional remote machine for monitoring and healing, it may be too expensive to be deployed in commodity uses.

Here we propose using self-virtualization to provide self-healing features to a computing system. As when activated, a VMM is in full control of the operating system thereon, the VMM is a good candidate to repair the tainted state of operating systems. Sensors could be added to monitor the anomaly of the operating systems. The sensors will trigger a self-virtualization of the operating system to partial-virtual mode, and the pre-cache VMM is activated to repair the tainted state. This approach is rather cheap in that it requires no additional hardware. Also, it incurs no performance degradation as the VMM is only required during system healing.

6.3 Online Hardware Maintenance

The market is heading toward 99.999% availability for IT infrastructures. Scheduled and unscheduled hardware maintenance could greatly disrupt the running system and significantly reduce the Mean Time to Interrupt (MTTI), thus lowering the availability of the IT infrastructure.

Self-virtualization supports online hardware maintenance with little or no performance penalty. In the framework of self-virtualization, an operator could switch the machine to be maintained to the full-virtual mode dynamically. The execution environment of the machine can then be live migrated to another machine that has been virtualized and is in the partial-virtual mode to accommodate multiple operating systems. After the maintenance work is completed, the execution environment is migrated back and the machine is returned to the native mode for full speed. Such online hardware maintenance could be accomplished transparently and seamlessly without the awareness of the running applications.

6.4 Live Updating Operating Systems

Many critical IT infrastructures require non-disruptive operations. However, the operating systems thereon are far from perfect that patches and upgrades are frequently needed to close vulnerabilities, add new features and enhance performance. To mitigate the loss of availability, such operating systems need to provide features such as live update[12,50] through which patches and upgrades can be applied without having to stop and reboot the operating system.

For example, one previous system called LUCOS[12] provides live update capability to Linux running on Xen. However, one drawback of LUCOS is that it requires a VMM permanently underneath the operating system to update. Self-virtualization could effectively solve this problem. When there is a need to perform a live update, a VMM could be dynamically attached and the operating systems could be turned into partial-virtual mode. The attached VMM then applies the live update and is detached when the live update is completed. As the VMM is completely inactive in native mode, upgrades to the VMM are more straightforward as if the upgrades are statically performed. Hence, self-virtualization could greatly eliminate unnecessary performance overhead incurred by virtualization in LUCOS.

6.5 Improving the Availability of HPC Clusters

As more high-performance computing (HPC) clusters are used in mission-critical and long-running applications, high-availability and failure recovery capabilities are becoming more important to HPC clusters. Three main factors contribute to the loss of availability on HPC clusters: software maintenance, hardware maintenance and an unexpected system failure. As illustrated in Subsections 6.3 and 6.4, software and hardware maintenance can be solved by self-virtualization. Here, we focus on how to survive system failures using self-virtualization on HPC clusters.

For high performance computing, there are usually some hardware monitors to monitor the temperature, fan speed, voltage, and power supplies in the system. These can be facilitated for hardware failure prediction[51]. When hardware errors are reported by the monitors, the operating system immediately virtualizes itself to the full-virtual mode and migrates itself to another healthy node, which in turn virtualizes itself simultaneously to the partial-virtual mode to accommodate the migrated operating system. With this approach, the running programs are completely shielded from the system failures, with no need to stop and restart. Further, this approach incurs little or no performance degradation during normal execution since the operating system could execute in the native mode when no error is detected.

7 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 100 Hz for all systems.

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

7.1 Experimental Setup

The experiments were conducted on a system equipped with a DELL SC 1420 machine with two 3.0 GHz Xeon processors, with 2 GB SDRAM, one Realtek r8169 Gigabit Ethernet NIC in 100 M LAN, and one single 73 G 10 k RPM SCSI disk, with 20 GB 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 Linux Enterprise edition 4 was used throughout. It is installed on ext3 file system. 900 000 KB of memory is given to each variant of Linux except the unprivileged domain (i.e., domainU). DomainU is configured with 870 000 KB of memory as it relies on Domain0 to complete device accesses. Therefore, we decreased 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 20 GB partition in the same disk with the ext3 file system as well. The disk was used in "raw mode", which is believed to have the best performance. We tested two modes of processors: UP mode, by allocating a single processor for each system; SMP mode, by allocating two processors for each system.

For application-level benchmarks, we present the performance results for the Open Source Database Benchmark suite (OSDB)[52], dbench[53], 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 build a Linux Kernel 2.6.16 with gcc-3.3.3. For micro-benchmarks, we measured the *lmbench* benchmark of version 3.0-a5[54], and reported the results for the OS-related parts for all six systems. For network performance, we used Iperf[55] 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 five times and each result is an average of them.

7.2 MicroBenchmark

Table 1 and Table 2 show the OS-related results of lmbench in uniprocessor mode and SMP mode. Our measurements indicate 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 65% and 55% performance loss in uniprocessor and SMP mode and while for process creation the result is 80% and 75%. It should be noted that due to the introduced locks and possible contentions, most of the operations in SMP mode are a bit expensive compared to those in UP mode.

Table 1. Lmbench Latency Results in Uniprocessor Mode (Time in μs)

Config.	N-L	M-N	X-0	M-V	X-U	M-U
Fork Process	98	114	482	490	470	471
Exec Process	372	404	1 233	1 232	1 211	1 220
Sh Process	1 203	1 337	2 977	2 996	2 936	2 931
Ctx (2 p/0 k)	1.64	2.49	5.10	5.41	5.04	5.06
Ctx (16 p/16 k)	2.73	3.91	6.76	7.28	6.54	6.45
Ctx(16 p/64 k)	10.30	12.77	15.73	16.27	15.77	15.97
Mmap LT	3 724	3 995	10 579	11 800	10 867	11 067
Prot Fault	0.61	0.63	0.97	1.17	1.04	1.11
Page Fault	1.22	1.48	3.09	3.18	3.03	3.10

Table 2. Lmbench Latency Results in SMP Mode (Time in μs)

Config.	N-L	M-N	X-0	M-V	X-U	M-U
Fork Process	128	148	509	523	501	501
Exec Process	449	501	1 353	1 386	1 335	1 349
Sh Process	1 444	1 585	3 359	3 435	3 222	3 319
Ctx (2 p/0 k)	2.31	3.07	5.16	5.61	5.11	5.14
Ctx (16 p/16 k)	2.91	4.15	7.16	7.27	6.83	7.02
Ctx (16 p/64 k)	11.03	12.40	16.17	16.77	16.10	16.60
Mmap LT	5 449	5 731	12 200	13 000	12 433	12 533
Prot Fault	0.70	0.74	1.13	1.20	1.15	1.18
Page Fault	1.64	1.89	3.45	3.67	3.39	3.46

Despite a number pointer indirection introduced by the virtualization objects when accessing virtualization-sensitive code and data, Mercury still only incurs negligible overhead compared to its counterparts. This confirms that the mechanisms of Mercury have little performance impact and Mercury could significantly eliminate the performance overhead associated with virtualization.

7.3 Overall Performance

Fig.3 depicts the overall performance of Mercury against native Linux and Xen-Linux in uniprocessor mode. For application level benchmarks, domain0 (X-0) shows 15% performance degradation for dbench, while domainU (X-U) incurs 5% performance improvement. Both domain0 and domainU incur about 9% for Linux kernel build, and more than 20% for OSDB-IR. One exception is for dbench, where domainU is with slightly better performance than domain0 and even Linux. This is probably because dbench is a throughput-oriented application and the splitted device mode could cache some data to avoid some expensive disk operations, though at the cost of possible inconsistency during crash[56]. For ping and Iperf benchmarks, the performance losses reach more than 20% and 40% for domain0, and 60% and 70% for domainU. As the Xen architecture has evolved dramatically in Xen2 and Xen3, the results are somewhat biased with the early results of Xen[30]. However, our results mostly conform to a recent measurement by Soltesz *et al.*[34].

By contrast, the performances of Mercury in its three modes (M-N, M-V and M-U) are 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.

Fig.4 depicts the overall performance of Mercury against native Linux and Xen-Linux on SMP machines. The evaluation on five application level benchmarks has the similar results in uniprocessor mode. The overhead in Mercury in the three modes is less than 2% compared to native Linux, domain0 and domainU.

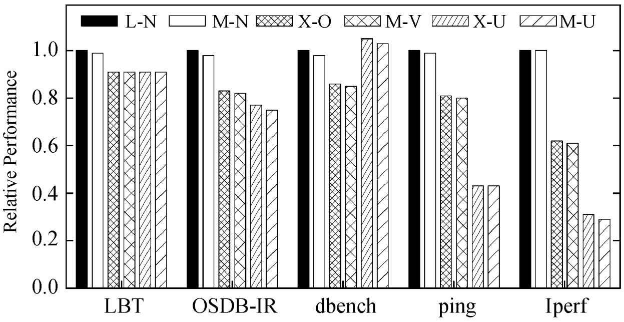


Fig.3. Relative performance of Mercury against Linux and Xen-Linux in uniprocessor mode.

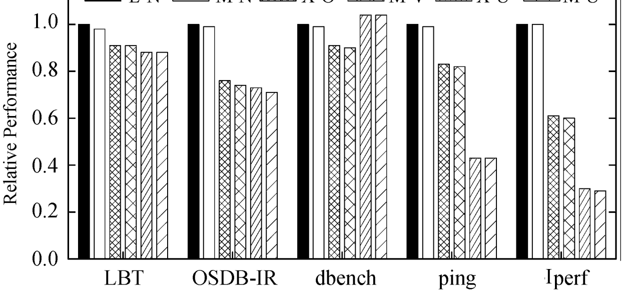


Fig.4. Relative performance of Mercury against Linux and Xen-Linux in SMP mode.

7.4 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. According to our tests, the time spent in a mode switch is relatively small: the average time is about 0.22 ms to do a switch from native mode to virtual mode, and 0.06 ms to a switch back.

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 Subsection 5.1.2, 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.

8 Conclusions and Future Work

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

Though Mercury has demonstrated the feasibility of embracing both performance and dependability in specific usage scenarios, there is still ample optimization and exploring space behind the implementation of Mercury. In the followings, we discuss several limitations and possible extensions to Mercury, which will be our future work.

Supporting Hardware-Assisted Virtualization. Currently, Mercury exploits the virtual machine interface (VMI) to make it easy to be adapted with the evolution of operating systems and VMMs. This still requires changes to the operating systems, though in a modular manner. Recent hardware advances have made hardware-assisted virtualization features commercially available. Hence, in our future work, we plan to extend Mercury to support hardware-assisted virtualization, by exploring existing hardware features, which could make the design and implementation of Mercury more clear and independent to OS evolutions. For example, current CPU virtualization such as VT-x enables the encapsulation of virtualization sensitive data into a centralized structure (e.g., VMCS or VMCB). This could make the mode switch between the native mode and virtualized mode much easier to implement. Further, the nested page table or extended page table could ease the tracking of the states of each page, which has been proved in our implementation as the most difficult port to debug.

Validating Mercury in Real Scenarios. Currently, we only validated Mercury in a Lab environment, where the resources are limited. This might not uncover many implementation limitations. For example, with the number of cores per-chip increasing continuously, the performance scalability of Mercury will be of great importance in supporting a relatively large-scale multicore machine. For example, a more loosely-coupled synchronization protocol might be necessary when detaching/attaching a VMM, instead of current protocols using IPI and shared variables. Further, we have not considered the case where the operating systems might have already been in an incorrect state during the mode switch. An OS not in a correct state might make the mode switch fail. Hence, a failure-resistant mode switch will be necessary to improve the dependability of Mercury itself.

References

[1] Goldberg R P. Survey of virtual machine research. *IEEE Computer*, 1974, 7(6): 34-45.

[2] Krsul I, Ganguly A, Zhang J, Fortes J A B, Figueiredo R J. VMPlants: Providing and managing virtual machine execution environments for grid computing. In *Proc. ACM/IEEE Conference on Supercomputing*, Pittsburgh, USA, Nov. 6-12, 2004.

[3] Adabala S, Chadha V, Chawla P *et al.* From virtualized resources to virtual computing grids: The In-VIGO system. *Future Generation Computer Systems*, 2005, 21(6): 896-909.

[4] Song Y, Wang H, Li Y, Feng B, Sun Y. Multi-tiered on-demand resource scheduling for vm-based data center. In *Proc. the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, Shanghai, China, May 18-21, 2009, pp.148-155.

[5] Zhang X, Dwarkadas S, Shen K. Hardware execution throttling for multi-core resource management. In *Proc. the 2009 Conference on USENIX Annual Technical Conference*, San Diego, USA, June 14-19, 2009.

[6] Sundararaj A I, Dinda P A. Towards virtual networks for virtual machine grid computing. In *Proc. the 3rd Virtual Machine Research and Technology Symposium*, San Jose, USA, May 6-7, 2004, pp.177-190.

[7] Dunlap G W, King S T, Cinar S, Basrai M A, Chen P M. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 2002, 36: 211-224.

[8] Joshi A, King S T, Dunlap G W, Chen P M. Detecting past and present intrusions through vulnerability-specific predicates. *ACM SIGOPS Operating Systems Review*, 2005, 39(5): 91-104.

[9] Chen H, Chen J, Mao W, Yan F. Daonity-Grid security from two levels of virtualization. *Information Security Technical Report*, 2007, 12(3): 123-138.

[10] Chen X, Garfinkel T, Lewis E C, Subrahmanyam P, Waldspurger C A, Boneh D, Dwoskin J, Ports D R K. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. the 13th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Seattle, USA, March 1-5, 2008, pp.2-13.

[11] Lowell D E, Saito Y, Samberg E J. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proc. the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, USA, October 9-13, 2004, pp.211-223.

[12] Chen H, Chen R, Zhang F, Zang B, Yew P C. Live updating operating systems using virtualization. In *Proc. the 2nd International Conference on Virtual Execution Environments*, Ottawa, Canada, June 14-16, 2006, pp.35-44.

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

[14] Youseff L, Wolski R, Gorda B, Krintz C. Paravirtualization for HPC Systems. Technical Report TR 2006-10, University of California, Santa Barbara, August 2006.

[15] Bjerke H K F. HPC Virtualization with Xen on Itanium [Master's thesis]. Norwegian University of Science and Technology, July 2005.

[16] Hu L, Jin H, Liao X, Xiong X, Liu H. Magnet: A novel scheduling policy for power reduction in cluster with virtual machines. In *Proc. IEEE Int. Conf. Cluster Computing*, Sukuba, Japan. Sept. 29-Oct. 1, 2008, pp.13-22.

[17] Chen H, Jin H, Shao Z, Yu K, Tian K. ClientVisor: Leverage COTS OS functionalities for power management in virtualized desktop environment. In *Proc. the 5th International Conference on Virtual Execution Environments*, Washington, USA, March 11-13, 2009, pp.131-140.

[18] Das T, Padala P, Padmanabhan V N, Ramjee R, Shin K G. Litegreen: Saving energy in networked desktops using virtualization. In *Proc. USENIX Annual Technical Conference*, Boston, USA, June 23-25, 2010.

[19] Ge R, Feng X, Song S, Chang H C, Li D, Cameron K W. PowerPack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*, 2010, 21(5): 658-671.

[20] Amazon Elastic Compute Cloud (Amazon EC2). Amazon Inc., http://aws.amazon.com/ec2/, 2008.

[21] Nurmi D, Wolski R, Grzegorczyk C, Obertelli G, Soman S, Youseff L, Zagorodnov D. The eucalyptus open-source cloud-computing system. In *Proc. the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, Shanghai, China, May 18-21, 2009, pp.124-131.

[22] Neiger G, Santoni A, Leung F, Rodgers D, Uhlig R. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel® Technology Journal*, 2006, 10(3): 167-177.

[23] Abramson D, Jackson J, Muthrasanallur S, Neiger G, Regnier G, Sankaran R, Schoinas I, Uhlig R, Vembu B, Wiegert J. Intel virtualization technology for directed I/O. *Intel® Technology Journal*, 2006, 10(3): 179-192.

[24] Bhargava R, Serebrin B, Spadini F, Manne S. Accelerating two-dimensional page walks for virtualized systems. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Seattle, USA, March 1-5, 2008, pp.26-35.

[25] Zhang X, Xiao L, Qu Y. Improving distributed workload performance by sharing both CPU and memory resources. In *Proc. International Conference on Distributed Computing Systems*, Taipei, China, April 2000, pp.233-241.

[26] Theurer A, Rister K, Krieger O, Harper R, Dobbelstein S. Virtual scalability: Charting the performance of Linux in a virtual world. In *Proc. Linux Symposium*, Ottawa, Canada, July 19-22, 2006, pp.393-402.

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

[28] Xiang S, Haibo C, Zang B. Characterizing the Performance and scalability of many-core applications on virtualized platforms. Technical Report FDUPPITR-2010-002, Parallel Processing Institute, Fudan University, November 2010.

[29] Edward W. Benchmarking Amazon EC2 for high-performance scientific computing. *Usenix Login*, 2008, 33(5): 18-24.

[30] Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A. Xen and the art of virtualization. In *Proc. the 19th ACM International Symposium on Operating System Principles*, Boston Landing, USA, October 19-22, 2003, pp.164-177.

[31] Schroeder B, Pinheiro E, Weber W D. DRAM errors in the wild: A large-scale field study. In *Proc. the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, Seattle, USA, June 2009, pp.193-204.

[32] Ford D, Labelle F, Popovici F I, Stokely M, Truong V A, Barroso L, Grimes C, Quinlan S. Availability in globally distributed storage systems. In *Proc. the 9th Usenix Conference on Operating System Design and Implementation*, Vancouver, Canada, October 4-6, 2010, pp.1-7.

[33] Huang W, Abali B, Panda D K. A case for high performance computing with virtual machines. In *Proc. the 20th Annual International Conference on Supercomputing*, Queensland, Australia, June 28-July 1, 2006, pp.125-134.

[34] Soltesz S, Pötzl H, Fiuczynski M E, Bavier A, Peterson L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proc. the 2nd European Conference on Computer Systems*, Lisbon, Portugal, March 21-23, 2007, pp.275-287.

[35] Vallee G, Naughton T, Ong H, Scott S L. Checkpoint/restart of virtual machines based on Xen. In *Proc. the High Availability and Performance Workshop*, Santa Fe, USA, October 2006, pp.1-6.

[36] VMware. The VMware software package. http://www.vmware.com, 2006.

[37] Russell R. x86 paravirt_ops: Binary patching infrastructure, http://lwn.net/Articles/194340/, 2006.

[38] Zachary A, Daniel A, Daniel H, Pratap S. Virtual machine interface (VMI). http://www.vmware.com/pdf/vmi_specs.pdf, March 2006.

[39] Clark C, Fraser K, Hand S *et al.* Live migration of virtual machines. In *Proc. the 2nd Usenix Conference on Networked System Design and Implementation*, Boston, USA, May 2-4, 2005, pp.273-286.

[40] Kappel J A, Velte A T, Velte T J. Microsoft Virtualization with Hyper-V. McGraw-Hill, 2009.

[41] Whitaker A, Shaw M, Gribble S D. Scale and performance in the Denali isolation kernel. In *Proc. Usenix Conference on Operating Systems Design and Implementation*, Boston, USA, December 2002, pp.195-209.

[42] KVM. KVM: Kernel-based virtual machine for Linux. http://www.linux-kvm.org/page/Main_Page, 2007.

[43] Intel Cooperation. Intel vanderpool technology for IA-32 processors (VT-x). http://www.intel.com/technology/computing/vptech/, 2005.

[44] Advanced Micro Devices. Secure virtual machine architecture reference manual. http://www.0x04.net/doc/amd/33047.pdf, 2005.

[45] Dong Y, Yang X, Li X, Li J, Tian K, Guan H. High performance network virtualization with SR-IOV. In *Proc. IEEE International Symposium on High Performance Computer Architecture*, Bangalore, India, January 9-14, 2010, pp.1-10.

[46] Sugerman J, Venkitachalam G, Lim B H. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proc. USENIX Annual Technical Conference*, Boston, USA, June 25-30, 2001, pp.1-14.

[47] Adams K, Agesen O. A comparison of software and hardware techniques for x86 virtualization. *SIGOPS Operating System Reviews*, December 2006, 40(5): 2-13.

[48] Kephart J O, Chess D M. The vision of autonomic computing. *IEEE Computer*, 2003, 36(1): 41-50.

[49] Bohra A, Neamtiu I, Gallard P, Sultan F, Iftode L. Remote repair of operating system state using Backdoors. In *Proc. Int. Conf. Autonomous Computing*, New York, USA, May 17-18, 2004, pp.256-263.

[50] Chen H, Yu J, Chen R, Zang B, Yew P C. Polus: A powerful live updating system. In *Proc. Int. Conf. Software Engineering*, Minneapolis, USA, May 20-26, 2007, pp.271-281.

[51] Leangsuksun C, Liu T, Rao T, Scott S L, Libby R. A failure predictive and policy-based high availability strategy for Linux high performance computing cluster. In *Proc. the 5th LCI International Conference on Linux Clusters: The HPC Revolution 2004*, Austin, USA, May 18-20, 2004, pp.18-20.

[52] Riebs A, Kirkwood M, Suchomski M, Eisentraut P, Clift J, Wagner P. The open source database benchmark, http://osdb.sourceforge.net.

[53] Tridgell A. Dbench filesystem benchmark. http://samba.org/ftp/tridge/dbench/.

[54] McVoy L, Staelin C. lmbench: Portable tools for performance analysis. In *Proc. USENIX Annual Technical Conference*, San Diego, USA, January 22-26, 1996, pp.279-294.

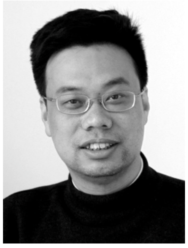
[55] Tirumala A, Qin F, Dugan J, Ferguson J, Gibbs K. Iperf: The TCP/UDP bandwidth measurement tool. http:// sourceforge.net/projects/iperf/, 2004.

[56] Yang J, Sar C, Engler D. Explode: A lightweight, general system for finding serious storage system errors. In *Proc. the 7th Usenix Symposium on Operating Systems Design and Implementation*, Seattle, USA, November 6-8, 2006, pp.131-146.

Hai-Bo Chen received the B.Sc. and Ph.D. degrees in computer science from Fudan University in 2004 and 2009, respectively. He is currently a professor in School of Software, Shanghai Jiaotong University, where he leads the system software group in the Institute of Parallel and Distributed Systems. His research interests are virtualization, system software and computer architecture. He is a member of China Computer Federation.

Feng-Zhe Zhang received the Ph.D. degree in computer science from Fudan University in 2011. His research interests include system architecture, operating system security, reliability and virtualization.

Rong Chen received the B.Sc., M.Sc. and Ph.D. degrees in computer science from Fudan University in 2004, 2007 and 2011, respectively. He is currently an engineer in Parallel Processing Institute, Fudan University. His research interests are operating system, virtualization and parallel programming model.



Bin-Yu Zang received the Ph.D. degree in computer science from Fudan University in 1999. He is currently a professor and the director of Parallel Processing Institute, Fudan University, and is a senior member of China Computer Federation. His research interests are compilers, computer architecture and systems software. His main research interests include operating systems and system software, compiler and system architecture.

Pen-Chung Yew received the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 1981. He is a distinguished research fellow in the Institute of Information Science, Academia Sinica and a professor of the Department of Computer Science and Engineering, University of Minnesota. His major research interests are multi-core architectures, compilation techniques and OS for multi-core embedded systems.