A Lightweight VMM on Many Core for High Performance Computing

Yuehua Dai Yong Qi Jianbao Ren Yi Shi Xiaoguang Wang Xuan Yu
Xi'an Jiaotong University, China
xjtudso@gmail.com, {qiy, shiyi}@mail.xjtu.edu.cn

Abstract

Traditional Virtual Machine Monitor (VMM) virtualizes some devices and instructions, which induces performance overhead to guest operating systems. Furthermore, the virtualization contributes a large amount of codes to VMM, which makes a VMM prone to bugs and vulnerabilities.

On the other hand, in cloud computing, cloud service provider configures virtual machines based on requirements which are specified by customers in advance. As resources in a multi-core server increase to more than adequate in the future, virtualization is not necessary although it provides convenience for cloud computing. Based on the above observations, this paper presents an alternative way for constructing a VMM: configuring a booting interface instead of virtualization technology. A lightweight virtual machine monitor - OSV is proposed based on this idea. OSV can host multiple full functional Linux kernels with little performance overhead. There are only 6 hyper-calls in OSV. The Linux running on top of OSV is intercepted only for the inter-processor interrupts. The resource isolation is implemented with hardware-assist virtualization. The resource sharing is controlled by distributed protocols embedded in current operating systems.

We implement a prototype of OSV on AMD Opteron processor based 32-core servers with SVM and cache-coherent NUMA architectures. OSV can host up to 8 Linux kernels on the server with less than 10 lines of code modifications to Linux kernel. OSV has about 8000 lines of code which can be easily tuned and debugged. The experiment results show that OSV VMM has 23.7% performance improvement compared with Xen VMM.

Categories and Subject Descriptors D.4.0 [*Operating System*]: General; D.4.8 [*Operating System*]: Performance

General Terms Design, Measurement, Performance

Keywords Multi-core, Cloud Computing, Virtualization, Hardware Assist Virtualization

1. Introduction

Multi-core processors are default for nowadays servers. Modern servers are configured with more and more cores and RAM [5]. In the future, the cores in a die will double steadily [17]. Based on this observation, in this paper, we introduce a new lightweight VMM architecture for future servers with redundant computing resources (CPU cores and RAM). The new architecture reduces performance overhead induced by traditional virtualization technology and also makes the VMM more reliable.

With the support of the virtualization, a server with limited resources can host multiple operating systems with good isolation. These operating systems can share same CPU core and RAM. The VMM dynamically schedules these operating systems based on their states, idle or active. However, the virtualization technology in traditional VMM also induces performance overhead to guest operating systems [8]. The virtualization of instructions, memory operations and emulation of devices are sources of the performance overhead. For different virtualization technologies, performance overhead is different. Figure 1 shows the total time of socket and close system calls. In this micro benchmark, a socket is created by *socket* system call and immediately closed by *close* system call. Total time of calling this system call pair 1 million times is measured. Para-virtualized Linux has the most significant latency. The interception operations for *syscall* instruction of Xen mainly contribute to the performance overhead. HVM Xen in this paper is based on AMD'SVM technology [6]. In HVM Xen, the *syscall* instruction

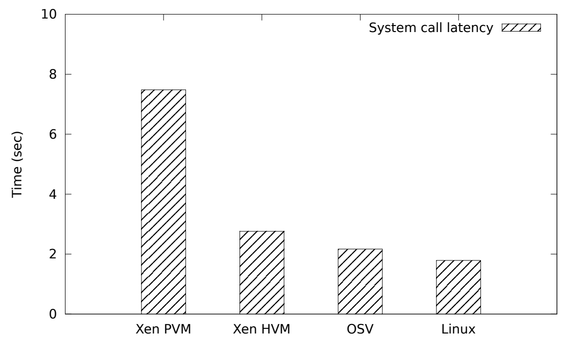

Figure 1. System call benchmark for native x86_64 Linux and Linux on Xen. The benchmark creates a socket by calling the socket system call, and then close the socket by calling system call of close on a AMD 32-core server. The time measured is the total time of calling the socket and close pair 1 million times. The para-virtualized Linux on Xen has the most significant latency. For x86_64 Linux, the system call is issued by *syscall* instruction, which is intercepted by Xen for para-virtualized Linux to prepare a faked system call frame for the guest Linux. The *syscall* instruction is not intercepted by HVM Xen which is based on AMD SVM technology, the performance overhead is mainly caused by the memory access overhead.

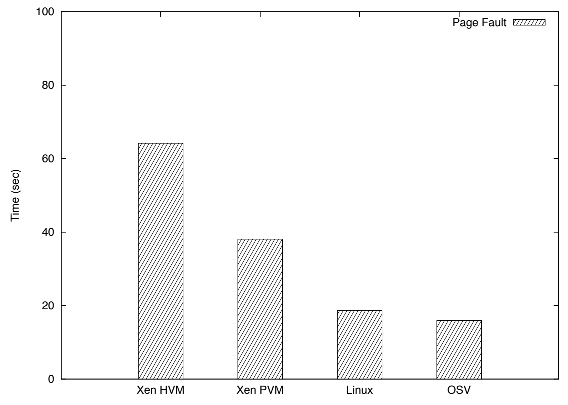

Figure 2. Memory management micro benchmark for native x86_64 Linux and Linux on Xen. Two threads allocate a 4KB page using *malloc*, then the page causing a page fault. The time measured is the total time of 1 million times. The HVM-Linux on Xen has the most significant latency. For x86_64 Linux, an extra nested page fault contributes to the latency. For para-virtualized Linux, the page is managed by the shadow page table, which is slower than native Linux. For OSV, the NUMA resource allocation makes it the low latency.

is not intercepted by Xen. The virtualization also contributes to the overhead of page fault handler of Linux. Figure 2 shows the total time of two threads touching 1GB memory in a 4KB page size fashion. The time for HVM Linux is larger than para-virtualized and native Linux. The extra time of HVM Linux is induced by nested page handler of Xen.

Virtualization layer is quite complex and makes whole VMM prone to bugs and security problems. For example, Xen has about 200K lines of code in the hypervisor itself, and over 1M in the host OS [19]. Some vulnerabilities and bugs in Xen have been reported [1–4]. Malicious users can attack Xen through these vulnerabilities, and take control over the whole system.

Nowadays, more and more companies are adopting VMM for their computing resource management. In a company, the employees are provided with thin computer clients which are connected to remote virtual machines allocated by VMM. The number of employees doesn't change so much in a period. The performance of virtual machines will affect efficience of employees. In this senario, virtualization provides convenience but is not necessary.

Besides, modern operating systems are capable of distributed services,for example, networked file system (NFS). Some devices can be exported as a service and shared by other operating system through standard distributed protocols. So, in a VMM, all devices can be managed by a privileged guest. The guest then exports these devices as a service to other guests through existing distributed protocols. In this way, devices can be shared between guest OS without virtualization. This can significantly simplify the VMM.

OSV is guided by the principle that guest operating systems should run directly on the server without the interceptions of VMM. This principle has two implications. First, the VMM should isolate operating systems from each other. Second, guest operating system manages its CPU core and RAM itself. The intended result is that CPU cores and RAM are not shared by guest operating systems. This limits the number of guest operating systems running on a server. However, as the number of CPU cores and RAM in a server increases, this is reasonable.

In this paper, we present an alternative approach - OSV, a light weight VMM, for constructing the VMM for these computers with number of cores. Rather than virtualizing resources in the computer, OSV only virtualizes the multiprocessor and memory configuration interfaces. Operating systems access the resources allocated by OSV directly without the intervention of the VMM, and OSV just controls which part of the resources are accessible to an operating system. OSV VMM also allows the operating systems to cooperate and share resources with each other, which makes the many core servers become a distributed system and keeps compatible with current computing base.The resulting system contains most of the features of traditional VMM, but with only a fraction of their complexity and implementation cost. This is quite suitable for the cloud computing.

In section 2, we provide our motivation for the OSV. Section 3 describes the design principles of OSV, while section 4 details the implementation. Section 5 evaluates the OSV and shows the experimental results. Section 6 summarizes related work. Limitations of OSV are discussed in section 7 and conclusion is given in section 8.

2. Motivation

The main goal of OSV is to simplify the design and implementation effort for constructing a VMM on multi-core servers. This section details why the design for OSV is reasonable on multi-core servers.

Virtual Machine Monitor(VMM) is prevalent in Cloud-Computing. By running numbers of operating systems concurrently, a single computer can satisfy various needs of applications at the same time, which makes the hardware running more efficient and reduces the hardware cost. Despite the convenience mentioned above, the VMM also incurs performance overhead. Performance overhead is varied with the types of application [8, 27]. The performance overhead is mainly caused by the virtualization of hardware resources and dynamic resource schedule policy. Virtualization and dynamic resource schedule are needed by a traditional VMM. This is because there are limited hardware resources in a computer such as CPUs, DRAM, I/O devices, etc, while operating systems use these resources in an exclusive fashion. Traditional VMM uses a privileged operating system to manage all the hardware resources. Other operating systems access the hardware resources through the privileged OS. In order to manage the hardware resources more efficiently, the VMM is becoming more and more complex.

In order to run the operating systems concurrently, traditional VMM must schedules the operating systems to make each operating system have the chance to get the CPU and do not conflict with each other, when there are a limited number of CPUs in a computer. The scheduler and virtualization are harmful to the system's performance and contribute large number of codes to VMM. For example, for a null system call the para-virtualized Linux one Xen needs 1042 cycles for an entry to the kernel and 476 cycles for returning to user space, while it is 148 and 134 cycles for the native Linux. That's why system call in PV guest is slow. On the other hand, as the number of cores and the amount of DRAM in a computer increase, the requirements for virtualizing the CPU and dram are not necessary. The maximum resources used by an operating system in a virtual machine is specified in a configuration file and is fixed when been created. The virtualization layer dynamically allocates the actual amount of physical resources to a virtual machine. For future multi-core servers, the physical resources is sufficient to fulfill all the guest operating systems. So, each guest operating system can be statically allocated with the amount of CPU cores and RAM as the configuration file specified.

The main functions of the virtualization layer are arbitrating access to memory, CPU, and devices, providing important network functionality, and controlling the execution of virtual machines [19]. Based on the cloud computing model and the major functions of VMM, using the static resource allocation policy is reasonable [24].

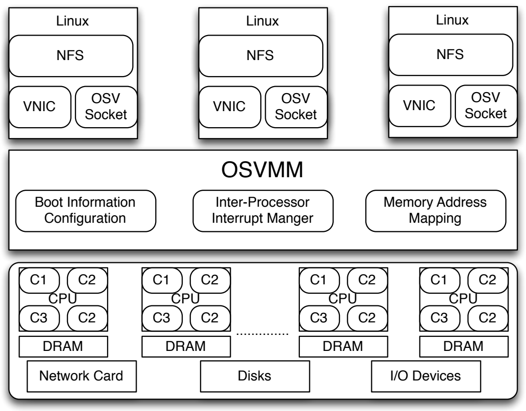

Figure 3. The architecture of OSV Kernel.

3. Design

The VMM poses several challenges for partitioning of a machine to support the concurrent execution of multiple operating systems. The most important one is that virtual machines must be isolated from each other. In a traditional VMM, VMM virtualizes or para-virtualizes the limited resources in a computer to isolate the virtual machines from affecting each other, but at the cost of increased complexity and reduced performance. This section details the design principle of OSV VMM. The intent is that the design principal will improve the guest operating systems' performance and simplify the VMM. The overall architecture of OSV kernel is shown in Figure 3.

3.1 CPU Cores

The architecture for future computers is far from clear, but the trend of increasing amounts for both cores and DRAM is evident. Commodity VMM virtualizes the CPUs and DRAM to run as many virtual machines as possible. This induces the performance overhead. We provide the virtual machines with physical CPU cores and DRAM without virtualization in OSV VMM. Each operating system running in a virtual machine manages the physical cores and DRAM itself without the intervention of the VMM. The operating system can only access the physical resources allocated to the virtual machine. The policy for CPU cores and DRAM allocated to a virtual machine is based on their NUMA node affinity. All of these approaches can promise the performance of the operating system.

The isolation for CPU cores between each virtual machine is guided by modifying the computers's processor information table such as ACPI or Multi-processor configuration table for Intel's Specification. OSV VMM masks processor entries in the table for the CPU cores not belonging to the virtual machine as unusable, which makes the operating system ignore these cores during booting period. The CPU cores allocated to the virtual machine are all in a NUMA node. By using the faked information, operating systems will only access the processors allocated to it. In the other hand, the VMM must guard all the procedure of SMP-booting, because the processors may be reset by the booting signal. The current implementation of OSV emulates the Intel's multi-processor specification [18] for SMP booting. The inter-processor interrupt(IPI) is widely used in the commodity operating system. When the IPI is issued by using the logical ID for a processor, the CPU cores in other virtual machines with the same logical ID will also get reaction to the IPI which will cause the virtual machine to break down. When the operating system initializes the CPU information and updates the APIC value, the VMM catches these updates and stores the corresponding physical ID for the logical ID of the CPU core. When sending the IPI, the VMM gets the physical ID from previous stored information and replaces the logical ID for sending the IPI. These operations need some minor changes to the source code of the operating system, for Linux, this just five lines of c code with inlined asm code. As the OS manages the processors itself, there is no need for OSV to intercept some privileged instructions such as *syscall* which improves the system call performance of the OS.The implementation details will be discussed in section 4.1.

3.2 Main Memory

The amounts of physical maim memory allocated to a virtual machine are as many as it requires. In this way, VMM can promise the performance of operating system. This approach requires more memory which can be easily satisfied by current servers. Traditional VMM uses a shadow page table to maintain the memory ranges accessed by an operating system. Furthermore, a VMM page fault handler for processing a physical page frame request from operating system should be provided. All of the above increase the complexity of traditional VMM and decrease the performance of operating system. Allocating as many amounts of physical meoroy to an operating system as it declares can avoid page faults in the VMM. This can improve the performance of the operating system and reduces the code size of VMM. Also, the operating system can access the main memory with real physical address which is helpful for DMA operations. When a device submit a DMA operation, it will use the exact page physical address to exchange the data. This can reduce the overhead of VMM for guaranteeing the isolation of the device operation in each operating system.

3.3 Devices

VMM nowadays is used to provide multiple special services in a server by running multi customized operating system instances. In this scenario, disks and network cards are the most used devices. In OSV, all devices are allocated to a privileged operating system. The I/O interrupts are all processed by it. The timer interrupt is forwarded to other virtual machines by sending an IPI to a CPU core of the virtual machine. Operating systems running on other virtual machines can share these devices via services exported through standard distributed protocol. There is a NFS server in the privileged operating system. All other virtual machines access the file systems via a NFS client which is widely used in nowadays commodity operating systems. All the file accesses are synchronized by the NFS protocol. There is no need for the guarantees of the VMM, which reduces the complexity of the VMM and improves the performance.

3.4 Virtual Network Interface

In order to allow the virtual machines to communicate each other, virtual network interfaces are provided by the VMM. The virtual machines can communicate with each other using standard distributed protocols over the virtual network interface. For example, the virtual machines share file through NFS. By the virtual network interface, traditional distributed protocol based applications can run on OSV VMM without code modification. This can keep compatible with current computing base.

There is a virtual network interface card(VNIC) for each virtual domain. Each VNIC has a private memory queue for receiving data packets. When the network code of the operating system submits a packet to the VNIC, the packet will be forwarded to the corresponding memory queue of the VNIC whose hardware address matches the packet's destination. When receiving a packet, the VNIC passes the packet to the network stack code. These memory operations for processing the packet are all done by the operating system without the interception of the VMM, which can increase the performance of the network stack. VNIC is provided as a driver module which can be installed in Linux without modifying Linux's source code.

3.5 Inter-OS Communication Socket

The performance of inter-OS communication via virtual network interface is limited by Linux's TCP/IP network stack. In order to increase the performance, a UNIX IPC like socket has been provided by the OSV. The API is the same as UNIX IPC socket. When creating a socket, one just needs to specify the socket type to OSV type. Then traditional socket functions such as listen, accept, send and so on are used to send and receive data. The data transmission mechanism is similar to virtual network interface: there is a memory queue for each virtual machine, each data packet is placed into proper queue by the destination virtual machine ID and the port. The memory operations are like the virtual network interface all done by operating systems which need not trap into the VMM.

As VNIC mentioned in 3.4, OSV socket is also a driver module installed in guest OS. Extra head files for C programmes are also provided. Applications with source code can be easily ported to this socket. By changing the socket type to OSV and using OSV defined address spaces, applications can communicate across guests via OSV socket. This is helpful for applications which are sensitive to the communication bandwidth.

3.6 Hypervisor Call

The design principle for the OSV kernel is that VMM restricts resource ranges accessed by operating system and lets the operating system to make the decision of resource usage. This principle reduces the interface complexity between operating system and VMM. In OSV VMM, it has only two types of hyper-call: resource access restrict functions and communication channel construction functions.

The resource restrict functions are used to isolate the resources allocated to each virtual machine and synchronize the accesses to shared resources. The resources are such as I/O devices, Local APIC, I/O APIC and so on. The later type of functions are used by the communication between operating systems. When two operating systems need to communicate with other, one operating system calls the function with the arguments of physical address of allocated memory and the two virtual machines ID pair, then the other operating system traps into the virtual machine to get the memory address with the ID pair. The shared memory and ID pair compose the communication channel. The memory queues for virtual network interface and inter-OS communication socket are constructed using these functions. The trap into VMM is only needed when the construction of the communication which can improve the performance of the communication and also reduce the complexity of OSV VMM.

3.7 Virtual Machine Construction

A virtual machine instance contains physical processor cores and main memory. Each virtual machine has statically allocated amounts of processor cores and main memory which can not be changed when constructed. The resources are allocated based on their NUMA node affinity. The main memory allocated are in the same NUMA domain with the processor cores which can reduce the memory access time. The I/O devices are only allocated to the privileged domain. Other domains can access the services provided by these devices through standard distributed services exported by the privileged domain. The OS image and its drivers are loaded into main memory when the system is booting.

The resources needed by a domain are preallocated before they are booted. A domain is created when the system load is high. The virtual machine instance is created by issuing a hypervisor call in privileged domain. When the virtual machine is booted, the tasks can be submitted to it using standard distributed protocols.

3.8 Isolation Between Guests

CPU cores and memory are preallocated to a guest OS. Guest OS initialises its internal structures based on these resources. So resources out of this range will not be accessed by the guest OS. OSV should control devices and interrupts to avoid guest OS interfering with each other.

For devices as described in previous section 3.3, all devices are allocated to a privileged operating system. So, interrupts from devices are not intercepted by OSV. The only thing OSV to do is to control port operations to avoid error states, such as reboot or reset operations. If a guest OS attempts to write a port, it will trap into OSV. OSV analyzes this operation to detect abnormal states, such as reboot operation, etc. If a reboot operation is found, OSV only reboots the corresponding guest OS not the computer. Operations which cause the server crash are denied by OSV.

IPI is a special interrupt used by operating system. OSV intercepts IPI to avoid a guest OS sending IPI to other guests. A miss-sending IPI will corrupt other guests. Based on these mechanisms, although there is no virtualization layer in OSV, guest OS can be well isolated from each other in OSV.

3.9 Application Programme Framework

A many core server running with OSV VMM is more like a distributed system. A map-reduce programming framework is provided in OSV. Applications written with the programming framework provided by the OSV are distributed among the OSes. The applications are submitted in the manager OS, and the results are also resembled in it. The framework is implemented on top of the OSV socket which can provides good performance.

4. Implementation

OSV VMM is implemented as a multithreaded program running on multi-core processor based servers. OSV differs from existing systems in that it pays more attention to the resources isolation rather than virtualize these resources. For example, OSV does not contain any structure to virtual the processor cores and main memory. The code size for OSV is about 8,000 lines of code which make it easier for tuning. The current implementation is based on AMD Opteron processors with SVM technology support and can run multiple Linux operating systems concurrently. The table 1 lists the approaches used by OSV to host multiple operating systems.

4.1 Multi-Processor Support

In order to support commodity SMP operating systems, traditional VMM needs a virtual CPU struct to provide IPI and schedules the mapping for the virtual CPUs and physical cores. In OSV, operating systems use physical CPUs directly. There are two challenges:

Multi-Processor Boot x86 based multi-processor systems are booted using universal SMP boot protocol. This protocol is to send two init inter-processor interrupts to the processor core. The init IPI will cause the processor to reset and jump to a specified location to execute. The reset action will make the processor's all the states be cleared including the registers initialized by the OSV. This will make OSV lose the control of CPU cores.

Inter-Processor Interrupt Traditional operating systems need the IPI to synchronize the system state and specify jobs for a CPU. Some IPIs are sent in logical destination mode which causes the CPU cores with the same logical id in other operating system making reaction to the IPI. This will lead the system to a unstable state.

Main Memory	
NUMA nodes	Each operating system can access the DRAM belongs to a NUMA node. The DRAM in other nodes are invisible to the OS. This is initialized in the E820 ram map.
Paging	For the privileged operating system, it works the same as in a bare metal. Other operating systems are controlled by a ncr3 register in AMD processors. Page size used in nested page table is splited, for low address 4KB size is used while 1GB for high address
Processor cores	
Multi processor	The processor cores allocates to an OS is in a NUMA node fashion. The cores in a NUMA node are allocated to an OS, which can reduce the remote cache access. The multi processor boot interrupt is redirected to a SX exception.
Interrupt	All the I/O interrupts are delivered to the privileged OS. Other OSes access the I/O through distributed protocols.
Timer	The external timer interrupt is dispatched by the privileged OS through the IPI.
Disks and I/O Devices	
Network Interface card	All the network cards are controlled by the privileged OS. An virtual network card is provided to other OS.
Disks, etc	These devices are exported as services by the privileged OS to other OS, which can be accessed through standard distributed protocols.

Table 1. The approaches for OSV to control resources for operating system

For the first one, we redirect the init interrupt to a Security Exception which will be caught by OSV VMM. In this way, the processor core can avoid being reset. When OSV catches the init IPI, OSV initialises the guest CPU mode to 16bit mode and gets the start code ip address for CPUs to execute after the init interrupt and redirects the CPU to this instruction when it returns from the VMM. The CPU will do as the traditional multiboot protocol does except for the reset action. The start code ip address is stored by Linux kernel which is 0x467. This address is specified by multi-boot specification. Current implementation of OSV is some tricky and hardly dependable for the AMD processors. Because init IPI redirection is based on the AMD's SVM technology. Modifying the source code of the OS is another way to support the multi-processor booting. The SMP-booting code is not so complex and irrelative for the performance of OS. For example, the SMP-booting code for Linux is all located in a C file. The modifications to the source code is less than 10 lines of c code, including init IPI sending functions and some port operations. This can make the VMM more portable.

For Inter-Processor Interrupts, some minor code modification should be made to operating systems as mentioned in section 3.1. OSV catches writes to some APIC registers. Normal APIC register operations are not intercepted by the OSV kernel except for IPI related registers. These registers are 0xD0 (Logical Destination Register) and 0x300 (Interrupt Command Register Low). OSV distinguishes logical and physical IPI destination mode through the APIC register 0x300. For physical mode, OSV lets it as the normal operating system does, while for logical mode, the kernel replaces the logical destination id with corresponding physical id and send the IPI with physical mode. The physical id for each CPU is unique which avoids CPUs in a domain making reaction to other domains IPI. This is lightweight compared with traditional VMM. The extra overhead for the APIC register interception is about 120-170 cycles for our servers. Compared to the latency of IPI interrupts, this is not critical to operating systems' performance.

4.2 OSV Socket

In cloud computing, operating systems running on a VMM communicate with each other frequently. They can communicate with each other through traditional network. In OSV, a socket is provided for improve the communication performance between operating systems running on OSV. The socket is UNIX IPC like. The data transfer between operating systems can be done without the intervention of the OSV kernel. This is about the implementation

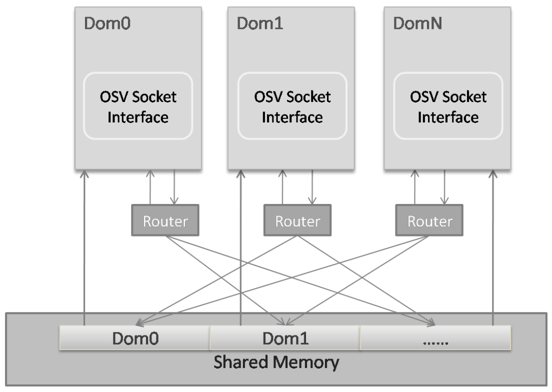

Figure 4. Structure for OSV socket.

of the osv-protocol. This protocol is based on shared memory and implements the Berkeley Sockets interface. Programmer could use this protocol as normal inet. Each domain in our OSV kernel is addressed by IP , and each program in a domain is distinguished by a access point, a port number. Figure 4 shows the overall structure of OSV socket.

There is only one receive buffer in each domain and one work queue is responsible for scanning the buffer periodically. In order to make domains be able to put message into each other's buffer, the kernel allocates the receive buffer in initialization and returns each buffer's physical address. For a socket, it has a buffer list to store the messages received but not handled. One socket is corresponding to one access point, distinguished by a unique integer.

There are two types of communication in osv protocol:

Inter-process communicate in a domain: The transfer procedure packages message into a osv message data structure which includes sender and receiver address, sender and receiver port number, message length, message data, message type. And then constructs a OSV skb, inserts it into the receiver's skb list. When receiver revokes a receive procedure, it will get this message form it's skb list.

Inter-process data transfer across domains: During the protocol initialization, each domain get the information about other domain's physical receive buffer address. So, they can put data in each other's receive buffer easily. Like IPC in a domain, the transfer packages the message firstly. After that, it puts the data into the corresponding domain's receive buffer according to the receive IP address. As each domain has a work queue scanning it's own receive buffer, the message can be discovered after being put into buffer immediately. And then, the message data is picked up from osv message data structure, packaged into a osv skb, inserted into the receiver's skb list. The remainder things will be done like IPC in a domain.

4.3 Linux Kernel

In order to host multiple Linux kernels concurrently, some modifications must be made to linux source code. Modifications to domain 0 and other domains are different:

Domain 0 The domain 0 needs to dispatch the external timer to other domains. So, in the domain 0 timer interrupt function should send IPI to other domains. This is done by issuing a hyper-call to *irq0_forward()*. The codes added to the linux kernel is just 5 line of inlined asm code.

Domain x The external timer interrupt for normal domains is received through IPI, so the timer interrupt should issue an EOI to the APIC of the CPU. So, the function of *ack_APIC_irq()* should be called in the timer interrupt function. Linux kernel assign each online CPU a logical id. When running multiple Linux, the logical id may be confused while sending IPIs. So assign of logical id to APIC should be intercepted by OSV kernel. And also, the action for sending IPI needs to be intercepted. The work is done by intercepting the writes to APIC registers. When the linux kernel writes the APIC register, it traps to the OSV kernel, the OSV translates the logical id to physical id and then send the IPI using the physical id.

All the modifications of Linux kernel is summarized in table 2. Total lines of code are less than 10, which is easily ported for linux.

5. Evaluation

This section demonstrates that the OSV's approach is beneficial to operating systems' performance. Performance of OSV is evaluated in this section.We first measure the overhead of virtualization using a set of operating system benchmarks. The performance is compared with Xen and native Linux kernel. Then the performance for the OSV's network system is measured. Finally, *memcached* is used to show the overall performance of OSV. The experiments were performed on two servers: Dell T605 two quad-core processors 2.0GHZ 2350 Opteron server with 16GB RAM, a Broadcom NetXtreme 5722 NIC and a 146GB 3.5-inch 15K RPM SAS Hard Drive, and Sun x4600M2 eight quad-core processors 2.8GHZ 8478 Opteron server with 256GB RAM and 1TB RAID0 Hard Drive. Linux version 2.6.31 was used, and compiled for architecture x86_64. The NFS version 4.1 was used. The opteron processors in both machines are with SVM and NPT support which are used by OSV. Both Xen and OSV guests are based on Linux kernel 2.6.31. The PV guest of Xen is configured without *superpage*.

5.1 Operating System Benchmarks

In order to measure the performance overhead of OSV VMM, we performed some experiments targeting particular subsystems. The lmbench [21] benchmark is used to measure the overhead. We compared the performance of Linux on bare metal, XEN HVM and PVM. The configuration for XEN guest OS is binded to a NUMA node: Guest OS can only access the drams and cpu cores on the specified node. This can reduce the schedule overhead of XEN and avoid cross NUMA node memory accesses. This expriment is carried on the 32-core server. For native Linux, lmbench is not configured to bind to a NUMA node. This is because for a 32-core

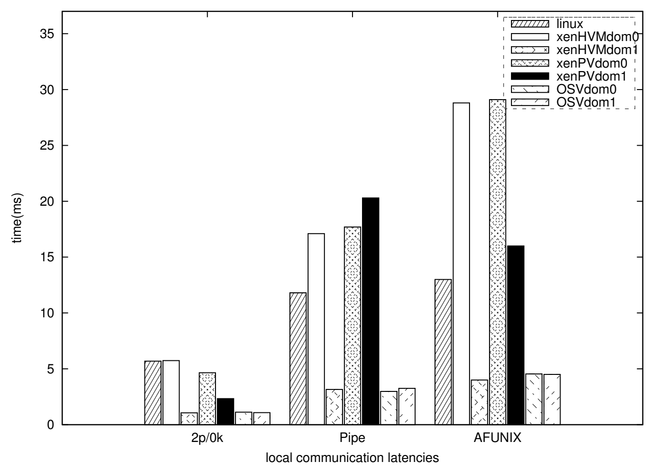

Figure 5. Local Communication Latency.

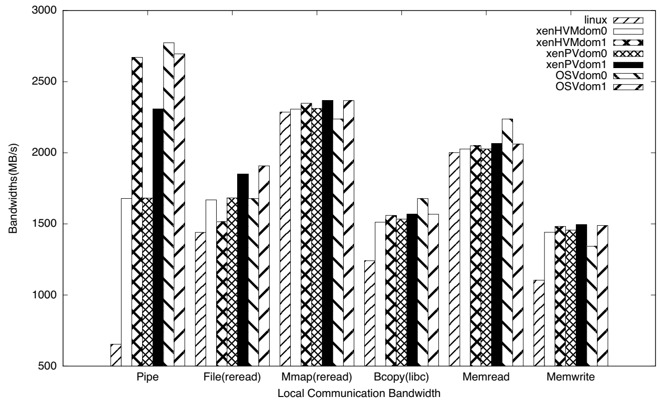

Figure 6. Local Communication Bandwidth.

server lmbench needs more memory than a NUMA node. Besides, lmbench also measures remote resource access latencies (including CPU cores and memory) which makes it should not be bound to a NUMA node. For OSV and Xen, guest OS kernels are configured with 4 cpu cores and 16GB drams.

The figure 5 shows the local communication latency. XEN HVM based guest OS gets the similar performance to OSV kernels and better than Linux and other PVM guests. This is mainly caused by the NUMA architecture of multi-core server: local access time is smaller than remote access. The resources used by the OSV kernel and XEN HVM is bound to a NUMA node so the cpu cores only have local dram accesses which make the performance better than Linux. The performance of XEN PVM guests is limited by the intervention of XEN when accessing the system resources.

Local communication bandwidth is shown in figure 6. The OSV kernel gets a high bandwidth compared to Linux and XEN HVM guest, especially in mem read test. The guest OS in OSV VMM accesses the resources allocated to it without the intervention of OSV VMM which makes it lower performance overhead. For the remote dram and cache coherent latencies, the Linux's bandwidth is the lowest with large number of cpu cores, 32-core in this experiment. In the File reread test, the OSV domain1 has the highest bandwidth. This is caused by the NFS based file system. When a file has already been accessed, it will remain in the NFS client cache, which

Domain 0	irq0_forward hyper-call is added to the timer interrupt function. 5 lines of inlined asm code.
Normal Domains	ack_APIC_irq() function call is added in the timer interrupt function; apic->write is replaced with osv_apic_write. 4 lines of c code.

Table 2. The modifications made to Linux kernel.

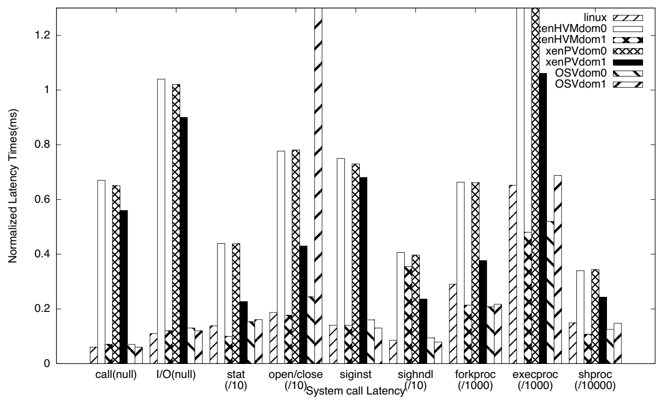

Figure 7. Processor and System call latency.

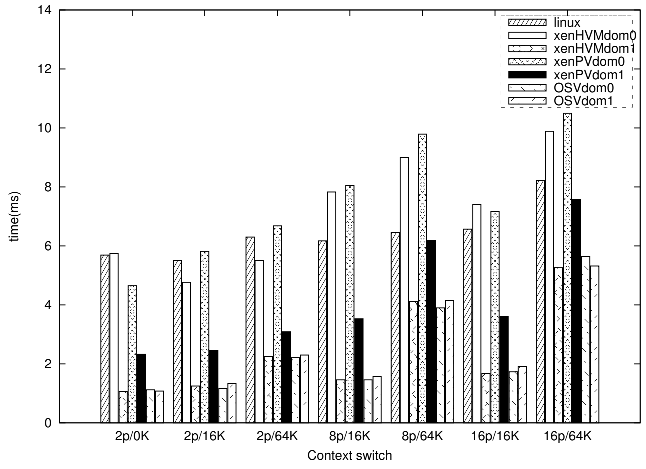

Figure 8. Context Switch Latency.

can accelerate the file read. The device of HVM guest on XEN is emulated, so the file reread bandwidth is lower then PVM guest.

In the context switch latency test shown in figure 8, the latency for OSV kernel is small which is similar to the HVM guest on XEN. The NUMA architecture and the TLB *shootdowns* contributes to the high latencies for Linux. The TLB *shootdowns* are expensive, when the number of cores is large. This contributes to the high latency. For PVM guest on XEN, each context switch needs the modification of shadow page table, so its latency is as high as the Linux. All the systems latency increases with the number of processes. The Xen domain 0 of both HVM and PVM have a high latency. This is because the domain 0 of XEN need to frequently be in service of the guest.

System call latency is critical to application's performance. The benchmark results are listed in figure 7. The Linux has the lowest latency for system calls, while OSV kernel and HVM guest on XEN get a similar result. The domain 0 for OSV kernel has the similar

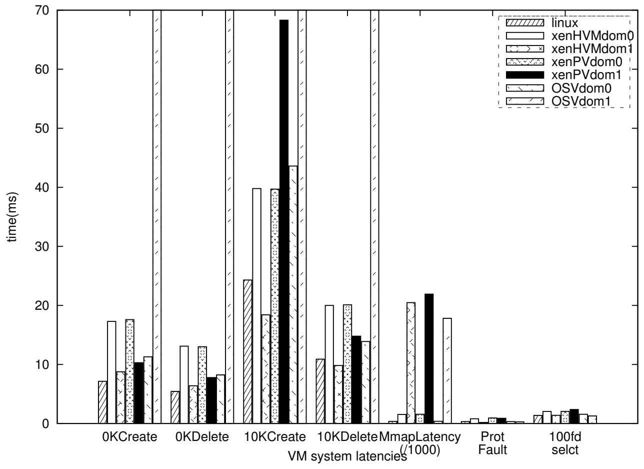

Figure 9. File and VM system latency.

performance compared to Linux. The domain 1 of OSV kernel in the open/close test get a super high latency. This is mainly caused by the NFS based File system. When opening and closing a file, the domain 1 needs two more network connection to finish the job which brings in the high latency. PVM guests on XEN have high latency in all the tests. These are all caused by the intervention of XEN when PVM performs some privileged operations.

The results for File and VM system tests which are shown in figure 9 are similar to the System call test. The domain 1 on OSV kernel has the biggest latency in tests except in prot fault and fd select. This also caused by the NFS filesystem. The network latency between domain 0 and domain 1 contributes the high latency. The domain 0 has a comparable performance to the Linux and XEN HVM guest. For mmap test, the domain 1 of XEN and OSV kernel all get a high latency, which is caused by the NFS file descriptor operations.

5.2 Socket Performance

In order to improve the communication performance of operating systems running on OSV, OSV socket is provided. We examine the bandwidth of the socket based on the TCP/IP over virtual network interface card and OSV socket protocol. The comparison is taken between Linux loopback and UNIX IPC which are widely used in the Linux system for SMP servers. We examine the time for transmitting 1GB data in different block size. The sender and receiver work in two different domains for OSV socket and VNIC. The VNIC is configured with 1500 MTU. For loopback test, the ip 127.0.0.1 is used. The UNIX IPC is measured between two processes. All the tests results are a median of 5 experiments.

The results are shown in table 3. When the block size is small the performance of OSV socket is comparable to UNIX IPC especially for the 256KB block size. This the same situation for the VNIC. For loopback card, when receiving a packet, it just passes the received sk_buffer to up layer by calling the function *netif_rx*. While works with VNIC, when receiving a packet from the packet buffer, the driver should make a call to *dev_alloc_skb* to get a sk_buffer for storing the received data, then passes it to

	OSV Socket	UNIX IPC	VNIC	lo
1K	1.82	1.59	4.73	2.32
8K	1.04	0.92	3.22	1.65
32K	0.93	0.77	2.75	1.3
128K	0.91	0.74	2.47	1.22
256K	0.87	0.74	2.51	1.16
1M	1.04	0.72	3.01	1.11

Table 3. Socket performance test, time in seconds. The test measures the time for transmitting 1GB data in different data block size.

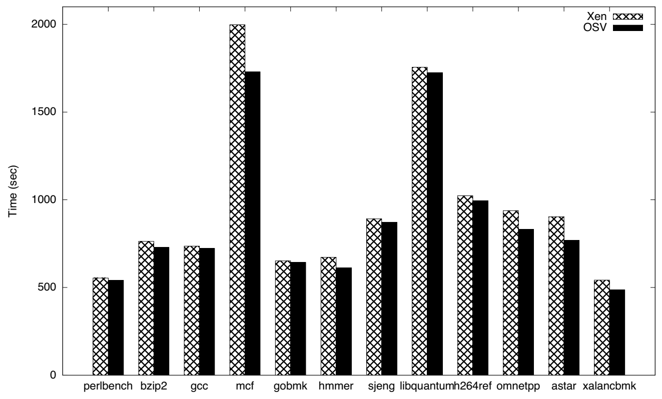

Figure 10. SPECint2006.

the up layer by calling the function *netif_rx*. This somewhat makes the performance for the VNIC worse than loopback. This is similar to the OSV socket and UNIX IPC. When the block size becomes large, the performance for VNIC and OSV socket decreases a little while the UNIX IPC and loopback card get good performance. This caused by the page frame management mechanism in Linux kernel. When the data block becomes large, it is difficult for the kernel to get continuous page frames to store these data from user space, this contributes mainly to the poor performance for the OSV socket and VNIC.

5.3 Application Performance

SPEC SPEC benchmarks are used to test the performance of the system under different workloads. We ran the benchmarks in both OSV and Xen with HVM Linux. Each VM was configured with four cores, 32GB of memory. The cores and memory allocated the a VM are in a NUMA node. There was only one VM running on the system. The device emulation was not used.

The results of our experiments are show in figure 10. We saw an approximately 2%-17% performance improvement across the board. The *astar* and *mcf* has the most significant performance improvement, which are 14% and 17%. *Mcf* and *astar* are memory heavy work, for example, *mcf* needs 1700 megabytes when running. The memory micro-benchmark results in figure 2 shows that OSV is efficient than Xen HVM. The performance improvement comes from the removal of the virtualization layer and the dedicated resource allocation policy.

Memcached Memcached is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering. Wikipedia, Youtube and Twitter all use memcached. We evaluated memcached in both OSV and Xen. The virtual machine configurations are the same as the SPEC experiments. Both Xen HVM and OSV are using

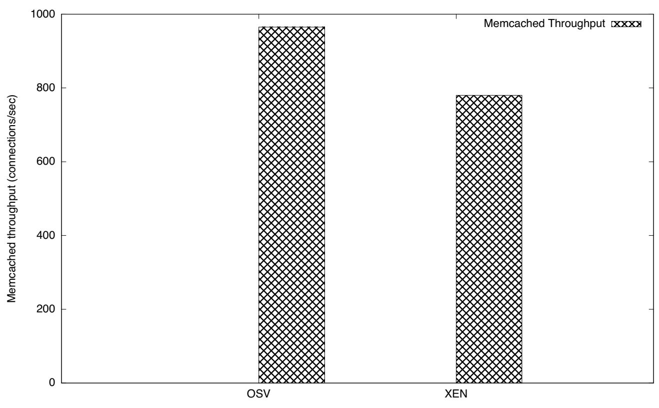

Figure 11. Memcached Throughput.

virtual network card. Physical network card in the privileged virtual machine is configured with port forwarding. Then, other computers can access memcached service provided by the virtual machine.

The connection throughput is measured. Results in figure 11 show that there is 23.7% performance improvement. For Xen HVM, the performance is limited by the frequently Nested Page Fault (NPT). HVM Linux needs frequently to trap into Xen. The context switch induces the performance overhead.

6. Related Work

Our work is inspired by many prior works. This section details the related work and summarizes them for differences between OSV VMM.

6.1 Virtual Machine Monitor

The VMM is used for hosting multiple operating system in a server for different application usage. OSV makes the many core system distributed which is similar to a VMM such as Xen [8], VMware [27]. But the OSV does not virtualize the resources and provides scalable performance while with little code base. No-hype [19] VMM is also focused on the removal of the virtualization layer from VMM. NoHype is focused on building a secure cloud VMM, while OSV is to improve the performance of VMM. In No-Hype, the hardware paging mechanisms are used to isolate each guest OS. For OSV, the hardware paging mechanisms only used in the boot time of guest OS, which is used to emulate some CMOS operations. After booted in OSV, the hardware paging mechanisms is turned off. So, OSV has lower performance overhead compared with Xen, while NoHype has a similar performance compared with Xen. OSV uses existing distributed services to share resources among guest OS. These services are based on a VNIC provided by OSV. NoHype is based on Xen and emulated devices are still needed.

6.2 Multicore OS

In order to make the operating system scalable on many core servers, researchers have done lots of work. K42 [7], Tornado [14] is developed in a object-oriented fashion. The kernel structures are controlled in distributed fashion, which makes the kernel scalable on many core platform. The multi-kernel [9] is similar to our work, it treats each core individually using message passing instead of shared memory to communication. Helios [22] is developed for heterogeneous architectures, it provides a seamless, single operating system abstraction across heterogeneous devices. These kernels scale well for multi-core servers, but the ABI(Application Binary Interface) is not compatible with current computing system. Disco [12, 16, 26] is a VMM developed to running multiple operating system concurrently to improve the scalability on multicore servers, which induces the virtualization overhead.

6.3 New Kernel

Exokernel [13] and Corey [10] are operating systems developed to provide the application more flexible resource management, which can reduce the overhead for resource contention and management overhead. Saha et al. have proposed the Multi-Core Run Time (McRT) for emerging desktop workloads [23] and explored configurations in which McRT runs on the bare metal, with an operating system running on separate cores. Salias et al have tuned the linux kernel for many core servers which can achieve the scalability for network applications [11], this needs the system specialized for an application. Libnuma [20] is proved by the linux kernel to improve the performance on NUMA based many core systems. Several studies with Linux on multicore processors [15, 25] have identified challenges in scaling existing operating systems to many core. All these work have inspired the work of OSV VMM.

7. Discussion

The motivation of implementing OSV is to build a neat and low performance overhead VMM. OSV tries to remove virtualization layer and uses a static resource allocation policy. Processor cores and memory ranges are pre-configured by OSV. So, OSV lacks the flexibility compared with traditional VMM. It is suitable for a cloud-computing model where users specify their resources demand in advance. Current implementation of OSV is based on AMD processors. It is easy to port OSV to other platforms with similar hardware assit virtualization technology. The only thing needs to pay attention to is the multi-processor boot trick used in OSV. However, this can be done by modifying the Linux source code which is described in section 4.1.

OSV demonstrates a way for constructing a lightweight VMM. The results above should be viewed as a case for the principle that VMM only controls the resources rather than a conclusive proof´. OSV lacks many features of commodity VMM, such as Xen, which influences experimental results both positively and negatively. Many of the ideas in OSV could be applied to existing VMM such as Xen. For example, booting guest OS with a predefined information table can be applied to Xen. Finally, it may be possible for Xen to provide a guest like the guest in OSV.

8. Conclusion

OSV is a lightweight VMM for many core servers. It provides the system scalability on many core servers, while keeps compatible with current computing base. It removes the virtualization layer in traditional VMM. Current distributed protocols are employed for resource sharing. TCP/IP and OSV socket are provided, by which the operating system can communicate with each other by standard distributed protocols, while by the OSV socket with high performance. The performance overhead for operating systems running in OSV is low compared with Xen VMM. The line of code for OSV kernel is about 8000, which make the system can be tuned for safety and reliability.

Acknowledgments

We would like to thank our shepherd, Leendert van Doorn, the anonymous reviewers, and Haibo Chen for their helpful suggestions for how to improve this paper. This research was supported in part by NSFC under Grant No.(60933003, 61272460), RFDP(20120201110010) and 863 Program (2012AA010904).

References

[1] CVE-2010-0633. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0633, 2010.

[2] CVE-2010-4255. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4255, 2010.

[3] CVE-2010-4247. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4247, 2010.

[4] CVE-2012-0217: SYSRET 64-bit operating system privilege escalation vulnerability on Intel CPU hardware. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0217, 2012.

[5] HP ProLiant DL980 G7 Server Data Sheet. `http://h20195.www2.hp.com/V2/GetPDF.aspx/4AA1-5671ENW.pdf`, 2012.

[6] AMD. Amd64 architecture programmer's manual volume 2: System programming. *Publication No. 24593*, September 2007.

[7] J. Appavoo, D. Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. Wisniewski, J. Xenidis, M. Stumm, et al. Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems (TOCS)*, 25(3):6–es, 2007. ISSN 0734-2071.

[8] 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. *Proceedings of the 19th ACM symposium on Operating systems principles*, pages 164–177, 2003.

[9] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Sch
"upbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles*, pages 29–44. ACM, 2009.

[10] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, et al. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 43–57. USENIX Association, 2008.

[11] S. Boyd-Wickizer, A. Clements, O. Mao, A. Pesterev, M. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010.

[12] E. Bugnion, S. Devine, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM SIGOPS Operating Systems Review*, 31(5):143–156, 1997. ISSN 0163-5980.

[13] D. Engler, M. Kaashoek, et al. Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review*, 29(5):251–266, 1995.

[14] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. *Operating systems review*, 33:87–100, 1998. ISSN 0163-5980.

[15] C. Gough, S. Siddha, and K. Chen. Kernel scalability—expanding the horizon beyond fine grain locks. In *Proceedings of the Linux Symposium*, pages 153–165, 2007.

[16] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 154–169. ACM, 1999. ISBN 1581131402.

[17] J. Held, J. Bautista, and S. Koehl. White paper from a few cores to many: A tera-scale computing research review.

[18] Intel. The MultiProcessor Specification Version 1.4. `http://download.intel.com/design/archives/processors/pro/docs/24201606.pdf`, 1997.

[19] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 350–361, New York, NY, USA, 2010. ACM. ISBN

978-1-4503-0053-7. doi: 10.1145/1815961.1816010. URL `http://doi.acm.org/10.1145/1815961.1816010`.

[20] A. Klein. An NUMA API for Linux. `http://www.firstfloor.org/andi/numa.html`, August 2004.

[21] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, page 23. Usenix Association, 1996.

[22] E. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234. ACM, 2009.

[23] B. Saha, A. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, et al. Enabling scalability and performance in a large scale CMP environment. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 73–86. ACM, 2007.

[24] J. Szefer, E. Keller, R. B. Lee, and J. Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 401–412, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0948-6. doi: 10.1145/2046707.2046754. URL `http://doi.acm.org/10.1145/2046707.2046754`.

[25] B. Veal and A. Foong. Performance scalability of a multi-core web server. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 57–66. ACM, 2007.

[26] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. *Operating system support for improving data locality on CC-NUMA compute servers*, volume 31. ACM, 1996. ISBN 0897917677.

[27] C. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):194, 2002.