# Improving Agility and Elasticity in Bare-metal Clouds

Yushi Omote
University of Tsukuba
omote@osss.cs.tsukuba.ac.jp

Takahiro Shinagawa
The University of Tokyo
shina@ecc.u-tokyo.ac.jp

Kazuhiko Kato
University of Tsukuba
kato@cs.tsukuba.ac.jp

## Abstract

Bare-metal clouds are an emerging infrastructure-as-a-service (IaaS) that leases physical machines (bare-metal instances) rather than virtual machines, allowing resource-intensive applications to have exclusive access to physical hardware. Unfortunately, bare-metal instances require time-consuming or OS-specific tasks for deployment due to the lack of virtualization layers, thereby sacrificing several beneficial features of traditional IaaS clouds such as agility, elasticity, and OS transparency. We present BMcast, an OS deployment system with a special-purpose *de-virtualizable* virtual machine monitor (VMM) that supports quick and OS-transparent startup of bare-metal instances. BMcast performs streaming OS deployment while allowing direct access to physical hardware from the guest OS, and then disappears after completing the deployment. Quick startup of instances improves agility and elasticity significantly, and OS transparency greatly simplifies management tasks for cloud customers. Experimental results have confirmed that BMcast initiated a bare-metal instance 8.6 times faster than image copying, and database performance on BMcast during streaming OS deployment was comparable to that on a state-of-the-art VMM without performing deployment. BMcast incurred zero overhead after de-virtualization.

***Categories and Subject Descriptors*** C.0 [*Computer Systems Organization*]: General—System architectures; D.4.7 [*Operating Systems*]: Organization and Design

***Keywords*** Operating Systems; Virtualization; Bare-metal Clouds; Device Mediators

## 1. Introduction

Most infrastructure-as-a-service (IaaS) providers today lease virtual machines (VMs) to their customers. Virtualization helps build their cloud infrastructures and provide certain essential characteristics of cloud computing, such as on-demand self-service, resource pooling, and rapid elasticity. Many research efforts have been devoted to improving the performance of VMs [17, 27, 39, 49], resulting in reasonable performance as a platform for most applications. However, virtualization could still become a significant obstacle to certain types of cutting-edge cloud applications that have significant performance, functionality, and security demands.

For example, big data applications, HPC applications, database servers, and media servers must avoid performance degradation and unpredictability caused by virtualization to maintain consistent high performance for uninterrupted real-time data processing [28, 29]. Google and Facebook do not use virtualization with their primary applications for this reason [2, 3]. In addition, certain applications require special hardware configurations, such as dedicated GPUs [8], InfiniBand network cards [10], specific SSD products [12], and RAID configurations [5]. Companies with strict security requirements hesitate to use virtualized multi-tenant environments [41, 50]. Therefore, a considerable number of today's customers demand physical machines (*bare-metal instances*) rather than VMs as their computing platform [28, 36]. To address this, several cloud providers, including IBM, have begun supporting bare-metal instances [4, 9, 10].

Unfortunately, bare-metal instances require either time-consuming or OS-specific tasks for the initial OS deployment. A straightforward approach to deploying an OS to an instance is to copy the entire OS image from a server to the local disk prior to starting-up the instance. Thus, the customer must wait for up to tens of minutes (depending on the network bandwidth and image size) for the copy procedure to finish. Rebooting the machine after the copy further increases the total wait time by several minutes, especially when the machine has a server-oriented motherboard with slow firmware initialization. This long deployment time significantly impairs the beneficial features of traditional IaaS clouds, such as agility and elasticity, and becomes an obstacle in, for instance, temporal testing, quick scale-up on demand, and hour-based pay-as-you-go services. The importance of startup time in VM instances has been recognized by both users and research communities [35, 40], and the same consideration applies to bare-metal instances.



*ASPLOS '15*, March 14–18, 2015, Istanbul, Turkey.

http://dx.doi.org/10.1145/2694344.2694349

OS streaming deployment [24] is a promising approach to reducing wait time. This approach first performs a network boot and then copies the OS image to the local disk in the background. This enables quick startup of instances and eventual bare-metal performance after OS deployment is completed. Unfortunately, OS streaming deployment depends heavily on OS functions and configurations, thereby sacrificing another important feature of the IaaS clouds, i.e., OS transparency. Abandoning OS transparency results in crucial limitations for not only cloud providers but also cloud customers because unskilled customers must test and verify the compatibility of special drivers with OS kernels whenever they update, patch, or customize their OSs. Providing pre-configured OS images may allow the customers to avoid such complications but significantly limits the choices of OSs the cloud customers can select. Using conventional virtual machine monitors (VMMs) provides OS transparency and mitigates the compatibility problems while allowing quick OS deployment [22]. However, even state-of-the-art VMMs incur continuous virtualization overhead and have difficulty achieving *pure* bare-metal performance [27, 32, 38, 45, 48].

Our study aims to incorporate several beneficial features of traditional IaaS clouds into bare-metal clouds: to improve agility and elasticity while preserving OS transparency without sacrificing performance. To this end, we introduce BMcast, an OS deployment system with a special-purpose *de-virtualizable* VMM that supports OS-transparent quick startup of bare-metal instances. BMcast performs streaming OS deployment transparently to produce a ready-to-use bare-metal instance quickly, while allowing direct hardware access from the deploying OS to minimize overhead as much as possible. After OS deployment is completed, BMcast turns off virtualization and disappears from underneath the deployed OS.

The key challenge of this approach is achieving two conflicting goals, i.e., sharing devices between the guest OS and VMM for low-overhead streaming OS deployment and seamlessly de-virtualizing devices for eventual bare-metal performance. Using virtual devices makes device sharing easy but makes de-virtualization complicated because device interfaces visible to the guest OS change during de-virtualization. Directly exposing physical devices to the guest OS makes de-virtualization easy but makes device sharing difficult. Although the basic concept of de-virtualization has been proposed previously [30, 34], runtime de-virtualization of I/O devices that are shared between the guest OS and VMM remains an open problem. To address this, we introduce *device mediators* that perform polling-based device-interface-level I/O mediation by carefully monitoring, intercepting, manipulating, and inserting I/O requests in a manner that conforms to device specifications. Device mediators allow physical devices to be first shared and then seamlessly de-virtualized.

The concept of device mediators is novel yet simple and generic. Developing device mediators requires the knowledge of device specifications to some extent; however, the task of device mediators is relatively straightforward, and therefore device mediators are much simpler and smaller than traditional device drivers. In fact, the device mediators we developed for IDE and AHCI devices are only 1,472 lines of code (LOC) and 2,285 LOC, respectively. We also found that MegaRAID SAS and Revo Drive PCIe SSD devices have similar straightforward interfaces. We believe that software interfaces of most storage host controllers have similarities because the types of underlying physical interfaces between the host controllers and disk drives are limited (SATA and SAS). Once implemented at the cloud provider side, device mediators simplify management tasks for customers significantly by providing OS transparency. More issues on the cost of device mediators are discussed in Section 6.

We implemented a prototype VMM of BMcast based on BitVisor [46]. We changed the core of BitVisor 1.4 by only 3,576 LOC. The prototype supports x86 environments with IDE and AHCI disk controllers, and Intel PRO/1000, X540, Realtek RTL816x, and Broadcom NetXtreme network interface controllers (NICs). We also designed and implemented a network storage protocol that extends the ATA-over-Ethernet (AoE) protocol [43] to improve network storage performance. Although sharing a NIC with the guest OS is technically possible, our current implementation uses a dedicated NIC for streaming OS deployment to avoid performance interference. We believe using a dedicated NIC for management is a reasonable configuration because modern server machines typically have multiple NICs. The possibility of sharing a NIC is discussed in Section 6.

Our current implementation does not yet fully support VMXOFF (i.e., completely turning off the CPU's hardware virtualization function). Although its performance benefit is negligible, supporting VMXOFF, in addition to supporting nested virtualization [20], would allow the guest OS to utilize the CPU's hardware virtualization function. Since supporting VMXOFF involves some complications, we currently only have an implementation with the support of a guest kernel module (not used in the performance evaluation). However, fully OS-transparent implementation in the VMM would be possible and is our future work.

The experimental results confirmed that BMcast can deploy Windows (Vista, 7, 8.1, Server 2008) and Linux (Ubuntu 10.04 and later, and CentOS 6.3 and later) without any modifications. BMcast started up a bare-metal instance 8.6 times faster than image copying, and the average database throughput on BMcast was comparable to that on a state-of-the-art VMM, i.e., kernel-based virtual machine (KVM) with exit-less interrupts (ELI) [1, 27] even though BMcast was performing streaming OS deployment while KVM with ELI was not performing that. After de-virtualization, BMcast incurred zero overhead.

The rest of this paper is organized as follows. Section 2 shows related work. Section 3 presents the design and Section 4 describes the implementation. Section 5 shows the results of a performance evaluation and Section 6 describes discussions. Section 7 draws the conclusion of this paper.

## 2. Related Work

We first present OS deployment approaches and then discuss VMM overhead from an OS deployment perspective.

***OS deployment:*** Image copying, such as in OpenStack Nova [7], is an OS-transparent but time-intensive approach to OS deployment. For example, copying a 30-GB image (default Windows Server 2008 on Amazon EC2) at 130 MB/sec (e.g., transfer from a 1.5 K rpm SAS disk over 10-Gb Ethernet) takes approximately 5 minutes. Using SSDs may reduce the copy time; however, the server or network may saturate when multiple instances are deployed simultaneously. Rebooting machines after the copy further add several minutes due to long firmware initialization time. Network installation, such as Kickstart in Linux, is a similar approach that copies and installs system files from the server to the local disk over the network. However, it is OS-specific and takes tens of minutes to complete the copy operation.

Network booting boots up an OS quickly; however, it does not deploy the OS image to a local disk, which causes continuous overhead as a result of redirecting every disk I/O over the network. Caching data on the local disk [22] reduces I/O overhead but must check cache expiration for every disk access on the critical I/O path, which increases disk access latency. OS streaming deployment [24] is a hybrid approach of network booting and image copying that allows fast OS startup and deployment to the local disk. Although it can reduce startup time and overhead, it still compromises OS transparency. Our approach enhances OS streaming deployment by exploiting a de-virtualizable VMM to provide OS transparency while retaining the low overhead during the OS deployment and eventual bare-metal performance.

***VMM overhead:*** Leveraging conventional VMMs, such as Xen [17] and KVM [31], is an easy approach to starting up an OS quickly while preserving OS transparency. However, despite efforts to reduce virtualization overhead, such as para-virtualization [17] and I/O pass-through [39, 49], such VMMs do not achieve bare-metal performance in compute-intensive and I/O-intensive workloads [18, 29, 37] due to the lock-holder preemption problem [47], cache pollution, nested paging, and interrupt handling overhead [21, 27].

Another possible approach is to uninstall VMMs after OS deployment. Several recent VMMs support raw disks and virtual-to-physical conversion. Therefore, using VMMs during streaming OS deployment and then converting virtual instances to physical instances is possible. However, uninstalling VMMs requires OS reboots, which incurs downtime. A recent study has shown that a more seamless conversion is possible by exploiting OS hibernation [32]. Unfortunately, in addition to slight modifications to the OS, this requires 90 seconds for physical-to-virtual conversion.

Creating virtual devices with the same device interfaces as those of physical devices will ease de-virtualization while preserving OS transparency. However, emulating an entire machine with the identical hardware interface as the underlying physical machine, including all devices (e.g., chipset and ACPI functions), is costly. In addition, synchronizing the internal states of virtual devices with physical devices is an open problem. Therefore, seamless and transparent conversion from virtual instances to physical instances is difficult.

NoHype [30] allows the removal of the virtualization layer to eliminate attack surfaces after booting guest OSs. Microvisor [34] demonstrates run-time de-virtualization for online maintenance of servers. Pass-through-based VMMs that allow direct control of I/O devices from the guest OS [44, 46] can also achieve de-virtualization. However, current de-virtualizable VMMs do not support device sharing between the guest OS and VMM. Therefore, the VMMs do not have access to the local disk to copy the OS image.

BMcast achieves seamless de-virtualization and eventual bare-metal performance, which differs from conventional VMMs. BMcast also achieves low-overhead device sharing with the guest OS, which differs from existing de-virtualizable or pass-through VMMs.

## 3. Design

In this section, we first explain the deployment process. We then describe the three BMcast functions, such as I/O mediation, background copy, and de-virtualization.

### 3.1 Deployment Process

BMcast deploys an OS by following four phases: initialization, deployment, de-virtualization, and bare-metal.

In the *initialization phase*, the VMM boots on a target machine. Although booting from the local disk is possible, we employ network booting to simplify VMM management (e.g., patching and updating). To allow fast startup, we minimize the VMM size as much as possible and optimize boot speed by parallelizing the initialization process. The VMM only initializes the dedicated NIC and leaves other hardware devices uninitialized because the guest OS will initialize the devices. The actual boot time is within a few seconds. In this phase, the local disk is uninitialized; all disk blocks are empty and contain no OS image (Figure 1a).

In the *deployment phase*, the VMM performs two tasks for streaming OS deployment: (1) copy-on-read to retrieve data from the server for empty blocks and (2) background copy to actively fill empty blocks. The copy-on-read allows quick startup of OS instances, and the background copy allows eventual bare-metal performance. In copy-on-read, the VMM retrieves data from the server if the blocks requested by the guest OS are empty. The VMM then returns the data

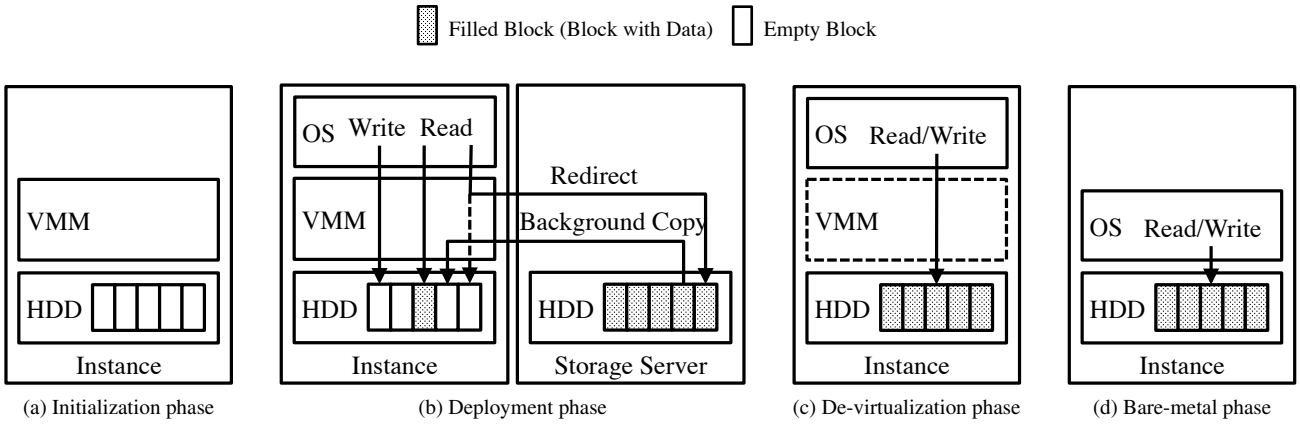



**Figure 1.** Four phases of deployment process

to the guest OS as if they are read from the local disk (see Redirect in Figure 1b). The VMM also writes the data to the local disk for future use. In other cases, I/O requests from the guest OS pass through the VMM and are served by the local disk (solid arrows from Read and Write in Figure 1b).

Parallel to copy-on-read, the VMM actively fills empty blocks with a background copy operation (see Background Copy in Figure 1b). The local disk is filled with the disk image gradually, and the guest OS eventually gains bare-metal disk performance. To reduce performance interference with the guest OS, the VMM moderates the speed of the background copy. The details of this process are described in Sections 3.2 and 3.3. To facilitate seamless de-virtualization, we use the identical block address space for both the local disk and the disk image on the server; the sector number of the disk image corresponds to that of the local disk.

In the *de-virtualization phase*, the VMM turns off virtualization to allow all hardware accesses to pass through the VMM (Figure 1c). To achieve seamless de-virtualization that is unnoticed by the guest OS, the VMM exposes identical hardware interfaces to the guest OS during de-virtualization, and the VMM waits for a consistent hardware state which is an appropriate time to stop virtualization. The details of this process are discussed in Section 3.4.

In the *bare-metal phase*, the VMM does not exist. The OS runs on the bare-metal instance directly (Figure 1d). At this point, the disk image is the same as that on the server, with the exception of already-written disk regions.

### 3.2 I/O Mediation by Device Mediators

Device mediators are the key component required to achieve device sharing with the guest OS while exposing the physical hardware interface to the guest OS directly. Device mediators perform I/O mediation; they mediate access to devices from the guest OS and VMM. I/O mediation involves three tasks: *I/O interpretation*, *I/O redirection*, and *I/O multiplexing*. I/O interpretation involves monitoring I/O sequences from the guest OS and determining their context. I/O redirection involves intercepting I/O requests from the guest OS and redirecting them to the server, and I/O multiplexing involves inserting I/O requests from the VMM to devices. Note that I/O interpretation is the basis of I/O redirection and I/O multiplexing. I/O redirection is used in the copy-on-read, and I/O multiplexing is used in the background copy.

***I/O interpretation:*** I/O interpretation determines the context of I/O sequences to properly control I/O access from the guest OS. For example, in disk controllers, device mediators interpret three types of contextual information: *command*, *status*, and *data*. The command information contains the operation type (e.g. read or write), logical block address (LBA), and sector count. The status information determines whether the device is idle or busy, which is required to determine when operations terminate. The data information is about the actual data transfer, such as the DMA buffer address of the guest OS.

Device mediators can easily capture this information by monitoring programmed I/Os (PIOs) and memory-mapped I/Os (MMIOs), in association with in-memory data structures including queues, based on device specifications. Although actual I/O sequences are device-specific, the basic concept can be applied to many different types of disk controllers. In addition, device mediators only need to determine these basic I/O sequences and can ignore other irrelevant sequences, such as device initialization and vendor-specific configurations. Therefore, device mediators are simpler and smaller than conventional full-spec device drivers.

***I/O redirection:*** I/O redirection is used in the deployment phase to redirect read access of empty blocks to the storage server. Device mediators must (1) capture the block information, (2) retrieve the corresponding data from the server, and then (3) pass the data to the guest OS.

Figure 2 illustrates the basic operations of I/O redirection. Device mediators use I/O interpretation to capture the command information (“1. Interpret” in Figure 2). By interpreting I/O requests, device mediators can determine whether an

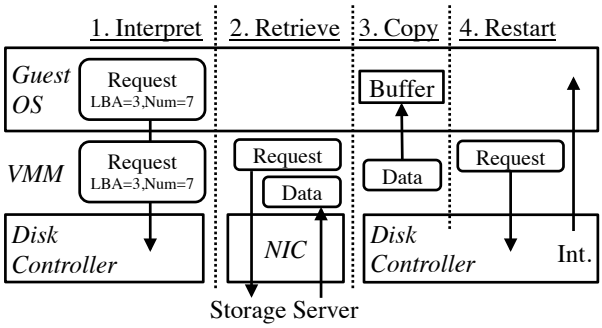



**Figure 2.** I/O redirection

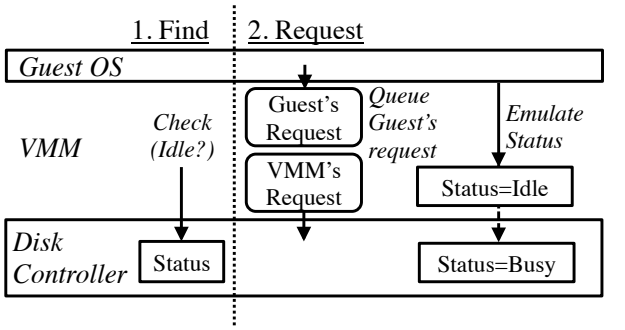



**Figure 3.** I/O multiplexing

access is read or not, and if read, its location (LBA and sector count). Based on this information, the VMM determines whether the blocks are empty or filled.

When blocks are empty, device mediators must retrieve the corresponding data from the server ("2. Retrieve" in Figure 2). First, device mediators temporarily block I/O access to the device so that the device does not start the data transfer. Device mediators then send the command information over the network and retrieve the data from the server. While retrieving data, device mediators emulate the status information so that the guest OS can determine that the device is busy. After retrieving the data, device mediators pass the data to the guest OS by functioning as a virtual DMA controller; i.e., copying the data to the guest DMA buffer ("3. Copy" in Figure 2). Device mediators obtain the address of the guest DMA buffer using I/O interpretation.

At this point, device mediators must generate an interrupt to indicate that the operation has been completed. The problem here is how to generate this interrupt. One possible approach is to virtualize interrupt controllers; however this complicates de-virtualization. Another possible approach is to share interrupt controllers with the guest OS transparently using techniques that are similar to I/O multiplexing, which is described below. However, sharing interrupt controllers is very complicated, especially when an advanced programmable interrupt controller (APIC) is used. Keeping track of interrupt numbers assigned by the guest OS to the devices is also difficult because it depends on the platform hardware such as low pin counter controllers and PCI bridges in the chipset. Therefore, this approach decreases portability drastically.

Rather than sending virtual interrupts, device mediators simply restart the blocked I/O access to the device so that the device itself generates an interrupt ("4. Restart" in Figure 2). To prevent the device from overwriting the guest DMA buffers, device mediators configure the device to transfer the data to dummy buffers. Device mediators also manipulate the command information (LBA and sector count) so that the device reads a single dummy sector that hits the disk cache. This technique allows easy and transparent generation of interrupts.

***I/O multiplexing:*** I/O multiplexing is used for background copy. I/O multiplexing allows the VMM to issue its own I/O requests to devices while the guest OS controls those devices. Device mediators must (1) find proper timings to insert I/O requests, (2) emulate device status and handle the requests, and (3) manage request queues.

Figure 3 illustrates the basic I/O multiplexing operations. First, device mediators find proper timing to safely share a device in a time-sharing manner ("1. Find" in Figure 3). If the device is processing an I/O request from the guest OS, the device mediators wait for completion of the request. Device mediators detect this using I/O interpretation.

When the request is completed, the device mediators send the I/O request from the VMM to the device ("2. Request" in Figure 3). To preserve consistency, device mediators emulate the status of the device as if the device is not busy even though the device is actually processing the request from the VMM. Therefore, at this time, the guest OS may attempt to send its own I/O request. To prevent conflicts with the guest OS, the device mediators intercept the request and keep it in a queue. After the request from the VMM is completed, the device mediators stop emulating device status and send queued requests to the device.

Sharing devices causes another problem on interrupts. Since obtaining interrupt numbers assigned by the guest OS is difficult, the device mediators cannot detect the exact interrupts generated for the requests from the VMM. In addition, the interrupts for the VMM are also delivered to the guest OS because interrupt controllers are not virtualized. Although device drivers may safely ignore unknown interrupts to support interrupt sharing, extra interrupt delivery should be avoided to preserve portability. Therefore, device mediators temporary disable interrupts and detect completion of requests by polling threads with fine-grained scheduling. The details of this process are described in Section 4.1.

### 3.3 Background Copy

To fill the entire local disk with the OS image, the VMM actively retrieves the data for empty blocks in the background. Since the retrieval and disk-write rates will differ, the VMM uses a pair of threads, *retriever thread* and *writer*

*thread*, that are connected by a FIFO queue. The retriever thread retrieves data for empty blocks from the server and pushes them to the queue. The writer thread then pops and writes the data to the local disk. The VMM fills blocks in order from low to high LBA. However, to minimize seek, the VMM changes the address adjacent to that of the last-accessed block if the guest OS accessed the disk.

In devices that have multiple request queues, a consistency problem may occur if a local disk is shared by the VMM and guest OS. To illustrate this problem, let us consider the following example. The VMM attempts to fill an empty block and send a request to the server. Before the response arrives, the guest OS issues a write request to the same block. In this case, the data from the guest OS is the most recent and should remain in the local disk. However, the response comes after the write from the guest OS; therefore, simply putting write requests into a queue in a FIFO manner breaks the consistency. To mitigate this problem, the VMM holds a bitmap to manage the status of each disk block and atomically checks the status to prevent the VMM from writing to a filled block. In case of shutdown and reboot, the VMM saves the bitmap on the local disk. The VMM uses an unused region on the local disk (such as unallocated space between two partitions) to save the bitmap. To protect the bitmap from the guest OS, the VMM prevents access to the region by converting it to read access to a dummy sector.

To avoid performance interference, the background copy speed should be moderated. If the write frequency is too high, guest storage performance degrades significantly. If it is too low, the deployment phase takes a long time. To solve this problem, the VMM adjusts the write frequency based on the guest OS load and three configurable parameters: *guest I/O frequency threshold*, *VMM-write interval*, and *VMM-write suspend interval*. If the disk I/O frequency becomes higher than the value of the guest I/O frequency threshold, the VMM waits for the time specified by the VMM-write suspend interval. Otherwise, the VMM writes blocks at the interval specified by the VMM-write interval.

Due to this moderation, the VMM will not perform excessive background copy operations during OS startup. As an optimization technique, we could configure the moderation function to prefetch the disk regions required for OS startup via a background copy operation, which would potentially boost OS startup time. However, we do not assume that this prefetch information is generally available.

### 3.4 De-virtualization

To make de-virtualization easy, we directly expose physical hardware to the guest OS. For example, all physical CPU cores and I/O devices, including PCI devices and interrupt controllers, and other hardware functionalities such as ACPI are exposed to the guest OS.

Memory address space is identity-mapped; a guest physical address is identical to the machine physical address. However, the VMM must allocate its own memory region and prevent the guest OS from accessing it. Therefore, the VMM reserves the memory region by manipulating the BIOS function such that the guest OS does not allocate the region. The VMM also uses nested paging to prevent the guest OS from accidentally corrupting the memory region.

In the de-virtualization phase, the VMM turns nested paging off to eliminate the paging overhead. At this time, the VMM needs to invalidate TLBs on all CPUs. Unfortunately, the VMM cannot use inter-processor interrupts (IPIs) for TLB shootdown because it does not manage interrupt controllers. Fortunately, page mapping is constant (always identity-mapped) over the life time of the VMM and does not cause a consistency problem among CPUs. Therefore, the VMM no longer needs to synchronize the timing of TLB invalidation for all CPUs. Each CPU can invalidate TLBs and turn nested paging off at different timings. After all CPUs disable nested paging, the VMM terminates virtualization.

## 4. Prototype Implementation

In this section, we describe our prototype implementation. First, we describe the CPU virtualization and network storage protocol. Then, we discuss the implementation status.

### 4.1 CPU Virtualization

We assume that the CPU is equipped with a hardware-assisted virtualization feature (Intel VT-x or AMD-V). To minimize overhead, we make the CPU run the guest OS as much as possible and switch to the VMM only when the minimum required events occur. The switch from the guest OS to the VMM is called *VM exits*.

We identified several events that require VM exits. To implement I/O mediation, PIO and MMIO instructions for the storage devices need to trigger VM exits. To detect boot, Startup IPIs and INIT signals need to trigger VM exits. To detect changes in paging and the processor mode of the guest OS, CR0 (PE, ET, WP, AM, NW, CD, PG) and CR4 (PSE, PAE, PGE, VMXE, SMXE, PCIDE, SMAP, SMEP) bit changes should also trigger VM exits. To trigger VM exits on MMIO, the VMM uses nested paging (EPT on Intel VT-x or NPT on AMD-V) and keeps the target memory regions unmapped. To cause other VM exits, the VMM configures the data structure to control the virtual machine (VMCS on Intel VT-x or VMCB on AMD-V). Note that the CPUID instruction unconditionally causes VM exits; however, this instruction occurs infrequently.

To poll devices in I/O mediation, the VMM needs to be scheduled periodically. With Intel VT-x, we exploit *preemption timer* to schedule threads. The preemption timer is supported by latest Intel CPUs that unconditionally causes VM exits at a specified interval. It allows fine-grained control of the timing of VM exits with the granularity of CPU clock cycles. Polling intervals are estimated from recent average network round trip times and I/O latency times. This achieves reasonable performance. If the preemption timer is not avail-

able, the VMM enables VM exits on hardware interrupts and uses a technique similar to soft timers [16].

### 4.2 Network Storage Protocol

The VMM requires a network storage protocol to redirect I/O requests to the server. To reduce overhead and improve transparency, we should select a protocol that allows the conversion of I/O requests to network packets with minimal effort. File-level protocols, such as NFS and CIFS, are therefore not suitable, and a block-level protocol is preferable.

We extended the AoE protocol [43], which has the greater affinity with ATA devices, although other protocols could be used. This protocol has a header that contains a field of device registers, and the VMM can easily convert the values of device registers to an AoE header. Data to be read is transferred as a payload of a packet with the header. If the transferred data is too long for a single Ethernet packet, the VMM splits the data into multiple AoE fragments. In this case, the VMM sets the tag field in an AoE header to determine the offset of a received fragment.

To improve performance, we modified the AoE protocol to support jumbo frames. We also add a retransmission capability to tolerate packet loss. We use *vblade* [15] as the basis of our AoE server implementation. However, the original vblade cannot fully utilize the network bandwidth because it is single-threaded and becomes a performance bottleneck when the VMM sends a significant volume of read requests. Therefore, we implemented a thread pool to vblade.

### 4.3 Implementation Status

We implemented a prototype VMM based on BitVisor [46]. It supports x86 environments with Intel VT-x or AMD-V CPUs. We implemented device mediators for IDE and AHCI disk controllers and the extended version of the AoE protocol in the VMM and server. We confirmed that the VMM can deploy both Windows (Vista, 7, 8.1, Server 2008) and Linux (Ubuntu 10.04 and later, and CentOS 6.3 and later) without any modifications to the OSs.

The sizes of device mediators are 1,472 LOC for IDE and 2,285 LOC for AHCI. We assume that we can use a dedicated NIC for streaming OS deployment to avoid performance interference. Therefore, we implemented small network drivers for Intel PRO/1000, X540, Realtek 816x, and Broadcom NetXtreme network adapters. The implementation cost is limited because we need minimal functions to send and receive packets with polling: the PRO/1000 driver has 718 LOC, X540 driver has 614 LOC, RTL816x has 757 LOC, and NetXtreme has 620 LOC. Our implementation is based on BitVisor 1.4 and modified the core of BitVisor by only 3,576 LOC. The total size of the VMM is approximately 27 KLOC. When adding device mediators for new devices, the VMM core does not need to be modified.

Our current implementation has some limitations. First, the memory region used for the VMM, currently 128 MB, is not released back to the guest OS after de-virtualization. We can mitigate this by implementing memory hot-plug features. Second, we hide hardware-assisted virtualization features from the guest OS because we have not implemented nested virtualization. However, nested virtualization is known to be implementable [20]. Third, we do not yet fully support VMXOFF (disabling the VMM mode).

To support VMXOFF, we need to perform the following operations on a VM exit: issue a VMXOFF instruction, restore the guest CPU state in memory (VMCS or VMCB) to the real CPU without using a VM enter operation, and jump to the instruction that caused the VM exit. These operations must be performed in the guest context since the VMM context was already lost on the VMXOFF. Therefore, we need memory pages mapped in the guest context to perform these operations. We confirmed that it is possible to implement this with the support of a guest kernel module (not used in the performance evaluation). However, it is theoretically possible to implement without a support of the guest OS by using, e.g., an extra page of device's PCI BAR (Base Address Register) [27]. Therefore, these limitations are not essential and we plan to implement them in the future version.

The NIC dedicated to the VMM is not explicitly released back to the guest OS. However, if the guest OS tries to detect it after de-virtualization, it can be found. If the dedicated NIC was connected to a management network and should not be exposed to the guest OS for the security reason, the VMM should continue to exist without performing VMXOFF and hide the PCI configuration space for that NIC. Its overhead will be negligible since a normal (non-malicious) guest OS will not try to access such a hidden NIC.

## 5. Performance Evaluation

In this section, we show the experimental results of evaluating the performance of BMcast. First, we show the result of measuring the OS startup time to demonstrate that BMcast achieved quick startup of bare-metal instances. Second, we explain the result of database benchmarks to demonstrate that BMcast achieved low-overhead streaming deployment and eventual bare-metal performance. Third, we show the result of an MPI benchmark to show the performance of BMcast in a cluster environment. Forth, we show the result of a kernel compile benchmark to illustrate the overall performance of BMcast. Fifth, we show the result of micro benchmarks to reveal the effects on threads, memory, storage and network performance. Finally, we show the behavior of the moderation in background copy.

We used a cluster of machines (originally used for HPC applications in practice), each of which was a FUJITSU PRIMERGY RX200 S6 with two Intel Xeon X5680 processors (3.33 GHz, 2 $\times$ 6 cores, hyper-threading disabled), 96-GB memory, a Mellanox MT26428 InfiniBand card (4X QDR), and a Seagate Constellation.2 ST9500620NS (500GB/7200 rpm) SATA hard drive. It also had two Intel 82575EM gigabit NICs, one of which was dedicated to the

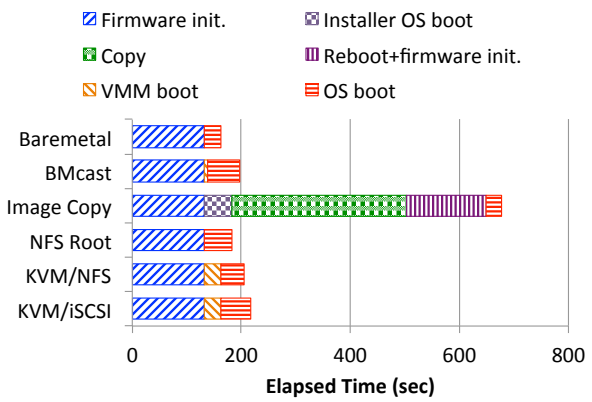



**Figure 4.** OS startup time

VMM. These machines were connected via a FUJITSU SR-S348TC1 gigabit Ethernet switch with 9000 bytes maximum transmission unit (MTU), and a Mellanox Grid Director 4036E InfiniBand switch. In all experiments, we deployed Ubuntu 14.04 with Linux kernel 3.13.0. For comparison, we used KVM (Linux kernel 3.9.0) with the ELI patch [1].

### 5.1 OS Startup Time

We first evaluate the OS startup time. We measured the startup time of an instance on both a bare-metal machine (Baremetal) and BMcast (BMcast). For comparison, we also measured the time of image copying over iSCSI (Image Copy). In addition, we measured the startup time of network boot using NFS (NFS Root), and that of a guest OS on KVM using a disk image over either NFS or iSCSI (KVM/NFS and KVM/iSCSI). In all experiments, we used gigabit Ethernet and a 32-GB OS image.

Figure 4 shows the results. The firmware initialization on our machine took 133 seconds, and the OS boot on the bare-metal machine took 29 seconds. On BMcast, the startup time of a bare-metal instance was 63 seconds (including 5 seconds to boot the VMM). On the other hand, image copying took 544 seconds (50 seconds to boot the installer OS over the network, 320 seconds to transfer the disk image, 145 seconds to restart, and 29 seconds for the OS boot from the local disk). Therefore, BMcast started up a bare-metal instance 8.6 times faster (excluding the first firmware initialization) or 3.5 times faster (including the first firmware initialization) than image copying.

The network transfer rate in the image copying was approximately 100 MB/sec, being limited by the bandwidth of gigabit Ethernet. Therefore, using SSDs will not speed up the image copying in this case. Using a faster network such as InfiniBand or 10Gbit Ethernet may reduce the transfer time. However, when multiple instances are started-up at the same time, the performance of the storage server will be saturated and the transfer time will not be reduced so much. On the other hand, BMcast transferred only 72MB of the disk image while booting the OS in 58 seconds, so the average rate was 1.2 MB/sec. This means that there is more room to scale-up the number of instances booted simultaneously. Therefore, BMcast will keep the advantage of fast startup time even if SSDs or faster networks are available.

In Figure 4, the OS startup time of network boot was 49 seconds, which was slightly faster than that on BMcast. However, it did not deploy the OS image to the local disk and took continuous overhead in a database workload. KVM took 30 seconds to boot itself. The BMcast VMM is designed to boot fast, as described in 3.1, and its boot time (5 seconds) was 6 times faster than that of KVM. The startup time of the guest OS on KVM was 42 seconds in the NFS case and 55 seconds in the iSCSI case. The startup time of the OS on BMcast, 58 seconds, was comparable to that in the iSCSI case. Moreover, when we compared the startup time including that of VMMs, BMcast was 1.14 times and 1.35 times faster than KVM with NFS and iSCSI.

From these results, we confirmed that BMcast achieved quick startup of bare-metal instances.

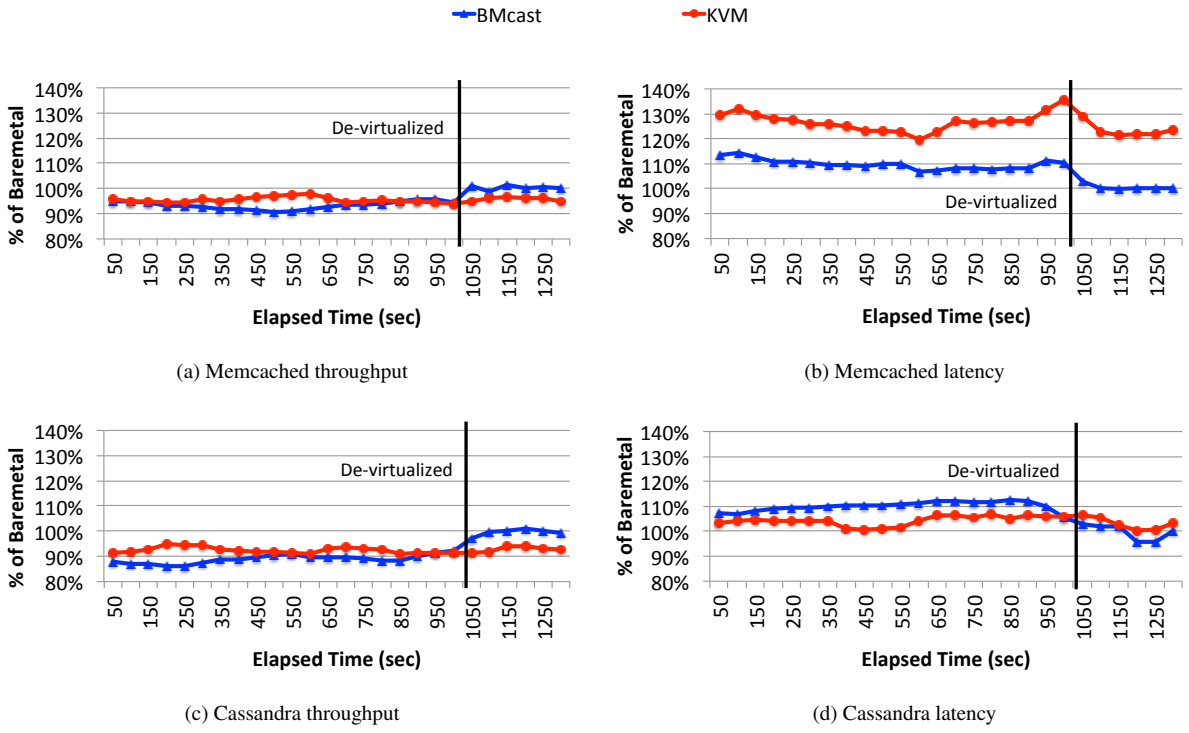
### 5.2 Database Benchmark

To evaluate the performance throughout the deployment and de-virtualization phase, we simulated a situation in which a user launches a new instance with a NoSQL database that serves data to clients, and traced the performance shift.

We tested two databases widely used in clouds: *memcached* and *Cassandra* [33]. Memcached is an in-memory database often deployed to improve data-serving latency for read-intensive workloads. Cassandra is a database that allows high throughput write access for update-intensive workloads. Both databases are designed to be scalable, spanning multiple instances depending on the amount of data to be handled. We used Yahoo! Cloud Serving Benchmark (YCSB) [23] for the performance evaluation that sends continuous requests from another instance to the database instance. For memcached, we recreated a read-intensive workload with 95% reads and 5% writes ratio and, for Cassandra, a write-intensive workload with 30% reads and 70% writes. We created a 32-GB OS image with the database configured.

Figure 5 shows the throughput and latency results of memcached and Cassandra benchmarks. The horizontal axis in each figure indicates the elapsed time from the beginning of the YCSB test. The vertical axis indicates the ratio to the average performance on the bare-metal machine. We performed the benchmarks on BMcast while streaming OS deployment was in progress and after de-virtualization (BMcast), and on a KVM instance with pass-through access to InfiniBand cards and para-virtual storage devices (KVM). Note that KVM did not perform streaming OS deployment and the overhead of sharing devices were not incurred.

In the memcached benchmark, even while BMcast was in the deployment phase and performing background copy, it showed slightly better performance than KVM. The average throughput was 34.6 kilo-transactions per second (KT/sec)




(a) Memcached throughput

(b) Memcached latency

(c) Cassandra throughput

(d) Cassandra latency

**Figure 5.** Throughput and latency of database benchmark

on BMcast, which was 94.8% of the bare-metal performance and 102% of the KVM performance (see Figure 5a). The average latency was 291 $\mu$sec on BMcast, which was 7% slower than the bare-metal performance and 14.8% faster than the KVM performance (see Figure 5b).

The primary reason for the performance degradation by BMcast is TLB pollution; the number of TLB misses increased up to 5 times and the latency on TLB misses doubled due to the two-dimensional page walks of nested paging. BMcast also consumed 6% of the total CPU time: 5% was for handling threads in OS streaming deployment and 1% was for the VMM core itself.

In the memcached benchmark, the deployment phase took 16 minutes. Therefore, in Figures 5a and 5b, the throughput increased and latency decreased after 990 seconds have elapsed; the throughput reached 36.4 KT/sec and the latency reduced to 281 $\mu$sec, which were identical to those of the bare-metal performance. There was no suspension or performance degradation during the phase shift. Therefore, we confirmed that BMcast achieved seamless devirtualization and eventual bare-metal performance.

In the deployment phase, the performance of the Cassandra benchmark on BMcast was slightly lower than that on the memcached benchmark. The average throughput was 51.4 KT/sec, which was 91.4% of the bare-metal performance and 98.7% of the KVM performance (see Figure 5c). The average latency was 2,609 $\mu$sec on BMcast, which was 7% slower than the bare-metal performance and 3% slower than the KVM performance (see Figure 5d). The deployment phase took 17 minutes (1020 seconds), which was longer than that in the memcached benchmark because the Cassandra benchmark was more write-intensive. However, after devirtualization, the throughput on BMcast increased to 60.0 KT/sec and latency decreased to 2,443 $\mu$sec, which were almost the same with that of the bare-metal performance.

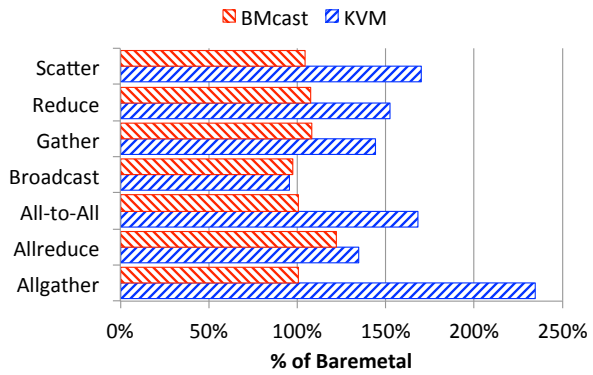



**Figure 6.** MPI benchmark

### 5.3 MPI Benchmark

To evaluate the performance effects of BMcast on cluster computing, we ran micro benchmarks of basic MPI operations. The cluster we used consists of 10 machines connected via an InfiniBand switch and OSs are configured to use

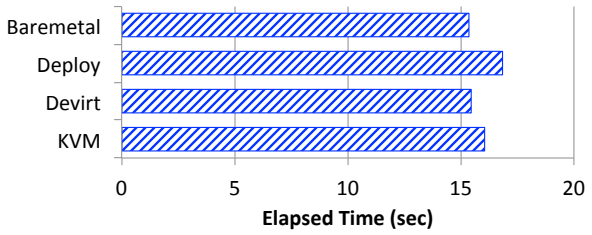



**Figure 7.** Kernel compile benchmark

MPICH2. We used OSU Micro-Benchmarks [6] and measured the latency of MPI collective communications among all machines. We ran the tests on the cluster with all OSs running on BMcast (BMcast). We also ran the same tests with all OSs running on KVM (KVM). We compared the result with that on the cluster of bare-metal machines.

Figure 6 shows the results. While most results on BMcast were almost the same with that on bare-metal machines, KVM incurred large overhead on many tests. On Allgather, the latency on KVM was 235% of that on bare-metal machines, while that on BMcast was almost identical to that on bare-metal machines. On Allreduce, BMcast incurred 22% overhead but KVM incurred 35% overhead. We guess the latency overhead of InfiniBand (see Section 5.5.3) largely contributes to this severe overhead on the MPI benchmarks.

### 5.4 Kernel-compile Benchmark

To illustrate the overall performance of BMcast, we used *kernbench*, which compiled a Linux kernel version 2.6.32 with the minimal configuration (`allnoconfig`) and 12 parallel jobs (`make -j 12`), and measured the total elapsed time on the bare-metal machine (Baremetal), on BMcast in the deployment phase (Deploy), on BMcast after de-virtualization (Devirt), and on KVM (KVM).

The benchmark results are shown in Figure 7. The kernel compile took approximately 16 seconds on the bare-metal machine. While OS deployment was in progress, BMcast increased the compile time by 8%. The main reason of this overhead was the cost of sharing the storage device between the guest OS and VMM by I/O multiplexing. However, its performance impact was limited because the moderation of background copy described in Section 3.3 was effective.

KVM (KVM) increased the compile time by 3%. This result does not include the cost of streaming OS deployment because KVM did not perform it. Therefore, this was pure virtualization overhead of KVM. After de-virtualization on BMcast, the compile time became identical to that on the bare-metal machine. These results show that the moderation of background copy is effective, and de-virtualization has the benefit of reducing the overhead of virtualization.

### 5.5 Micro Benchmarks

We performed three micro benchmarks to measure threads, memory, storage and network performance.

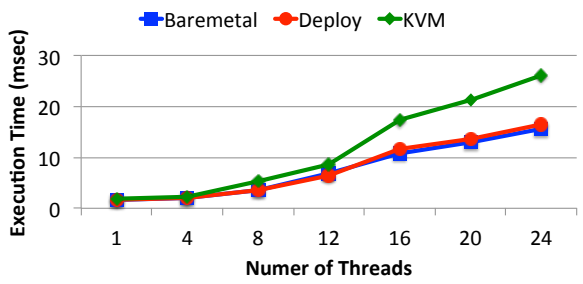



**Figure 8.** Threads performance

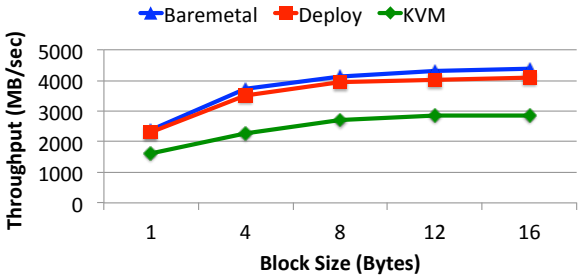



**Figure 9.** Memory performance

#### 5.5.1 Thread and Memory Benchmark

To evaluate the effect on threads and memory performance, we used SysBench [11]. The thread benchmark repeatedly performed a sequence of acquire-yield-release operations 1,000 times on 8 mutex locks from multiple threads and measures the average execution time. We changed the number of threads from 1 to 24. The memory benchmark repeatedly allocated a memory block and wrote data to the block until the total amount of data written reached 1 MB. We changed the size of memory block from 1 KB to 16 KB. We performed the benchmarks on the bare-metal machine (Baremetal), on BMcast in the deployment phase (Deploy), and on KVM (KVM). We configured KVM to use processor pinning to avoid overhead of scheduling virtual processors, and 2-GB huge paging to reduce overhead of nested paging.

Figure 8 shows the results of the thread benchmark. As the number of threads increased, the overhead of KVM significantly increased (68% on 24 threads). We guess this overhead was incurred by the lock-holder preemption problem [47]; a thread holding a lock was scheduled off from a virtual CPU and other threads must wait until the thread was scheduled back. This virtualization overhead could become a significant problem for highly-concurrent applications. On the other hand, BMcast incurred only moderate overhead (6% on 24 threads) even while streaming OS deployment was in progress. The reason of this was that BMcast traps only minimum events required for streaming OS deployment, and the frequency of VM exits were much lower than conventional VMMs.

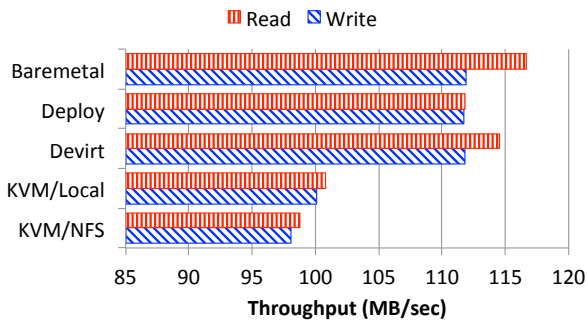



**Figure 10.** Storage throughput

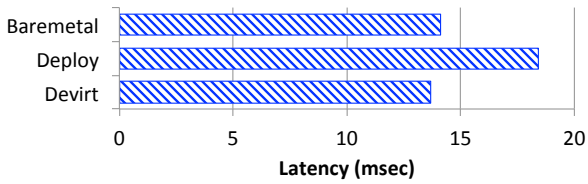



**Figure 11.** Storage latency

Figure 9 shows the results of the memory benchmark. While the throughput on BMcast incurred only 6% of overhead, KVM incurred 35% of overhead on 16-KB block size. We guess this overhead was caused by the cost of nested paging and cache pollution by the VMM including the host OS. These results confirmed that BMcast incurred low overhead on threads and memory performance during deployment. Note that after de-virtualization, the performance on BMcast became identical to the bare-metal case in both benchmarks.

### 5.5.2 Storage Benchmark

To evaluate the effects on storage performance, we measured the throughput and latency of disk access on the guest OS. To measure storage throughput, we used Flexible IO Tester (fio) [13] and read 200-MB of data with 1-MB block size using direct I/O and Linux native asynchronous I/O engine (libaio). To measure storage latency, we used ioping [14] and read 1 MB of data 100 times with 4K byte block size. We measured the performance on the bare-metal machine (Baremetal), on BMcast in the deployment phase (Deploy), and on BMcast after the de-virtualization (Devirt). For comparison, we also measured the performance on a network-booted OS (Netboot), on the guest OS on KVM with local disk (KVM/Local), and on that with NFS (KVM/NFS).

Figure 10 shows read & write throughput. On the bare-metal machine, the read and write throughput was 116.6 MB/sec and 111.9 MB/sec, respectively. The read throughput decreased by 4.1% in the Deploy case and 1.7% in the Devirt case. The write throughput was almost the same as the bare-metal case. This result suggests that the moderation of the speed in background copy worked effectively and

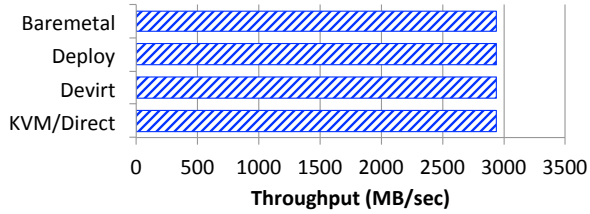



**Figure 12.** InfiniBand throughput

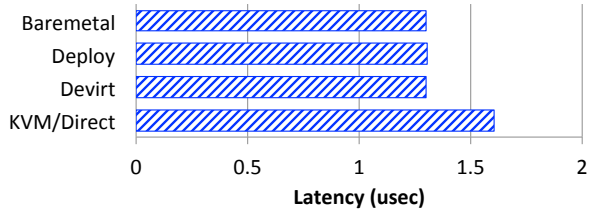



**Figure 13.** InfiniBand latency

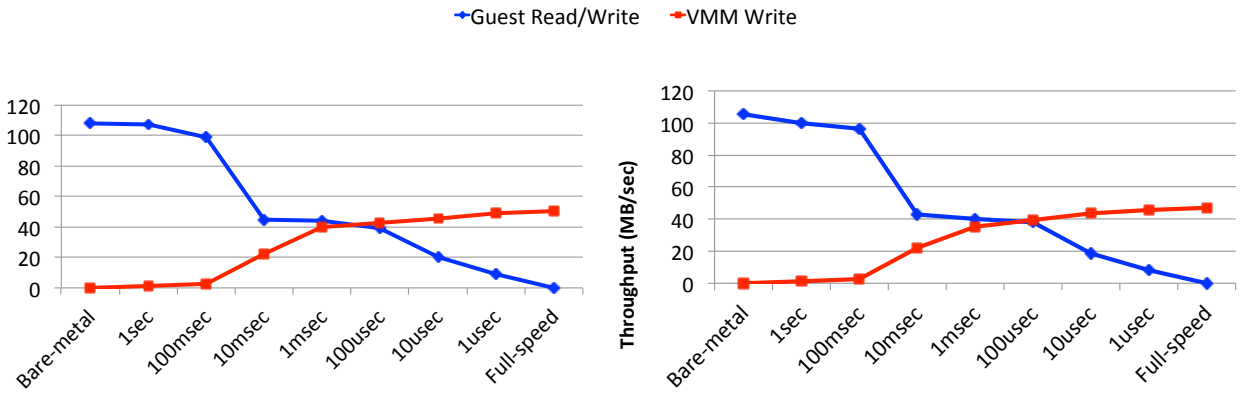
de-virtualization achieved almost bare-metal performance. On the other hand, KVM decreased the read throughput by 10.5% in KVM/Local and 12.3% in KVM/NFS, and decreased write throughput by 13.6% in KVM/Local and 15.3% in KVM/NFS, respectively. These overhead would be mainly caused by virtual I/O devices.

Figure 11 shows storage latency. BMcast in the Deploy case increased the average latency by 4.3ms. This increase was for the blocking time in accessing storage devices. As described in Section 3.2, if I/O requests from the VMM are being handled, the requests from the guest OS are queued and blocked. This blocking time was measured as the latency overhead. However, in the Devirt case, there was no latency overhead (actually slightly faster due to a mysterious behavior). In this case, no instructions, except for CPUID, triggered VM exits. The intervals of the CPUID exits ranged from a couple of seconds to minutes, and their overhead was negligible. These results confirmed that BMcast achieved eventual bare-metal storage performance.

### 5.5.3 Network Benchmark

To evaluate the effects on network performance, we measured the raw InfiniBand throughput and latency. We used the *ib_rdma_bw* and *ib_rdma_lat* commands in the *perftest* package of the Open Fabrics Enterprise Distribution. These commands sent 64-KB packet 1,000 times and measured the throughput and latency of remote direct memory access (RDMA) over InfiniBand. We measured the performance on the bare-metal machine (Baremetal), on BMcast in the deployment phase (Deploy), on BMcast after de-virtualization (Devirt), and on KVM with direct device assignment (KVM/Direct).

Figure 12 shows the throughput and Figure 13 shows the latency. In our environment, there was no difference




(a) Guest-read and VMM-write I/O throughput

(b) Guest-write and VMM-write I/O throughput

**Figure 14.** Guest and VMM I/O throughput with changing the intervals of the VMM writes

in throughput. At this time, CPU utilization was very low. This means that network was saturated, and the virtualization overhead was hidden by the command queuing of the RDMA hardware. However, KVM increased the latency by 23.6%. The cause of this would be the overhead of Intel's input/output memory management unit (IOMMU), cache pollution, and nested paging. This overhead become a significant problem in latency-sensitive applications. On the other hand, BMcast incurred negligible overhead (less than 1%). Therefore, these results confirmed that BMcast achieved bare-metal performance in the InfiniBand environment.

### 5.6 Moderation of Background Copy

In the deployment phase, BMcast moderates the speed of background copy by adjusting the interval between each block write. To clarify the relationship between the interval and the storage performance of the guest OS, we measured read / write throughput of the guest OS and write throughput of the VMM with changing the intervals of the VMM writes. The block size written by the VMM was 1024 KB.

Figure 14a shows the relationship between read throughput of the guest OS and write throughput of the VMM. Figure 14b shows the relationship between write throughput of the guest OS and that of the VMM. On the both graphs, the left-most bar shows the bare-metal throughput (Bare-metal). We reduced the intervals of VMM writes from 1 sec to 1 $\mu$sec exponentially, and finally the VMM issued write requests without intervals (Full-speed).

As the intervals of VMM writes were reduced, the guest throughput gradually decreased, and the VMM write throughput gradually increased. The sum of both throughput did not reach the bare-metal throughput in either case because the VMM used a polling-based storage access, and the guest OS and VMM wrote different regions of the disk that increased seek overhead. Overall, however, the total performance was fairly reasonable, showing that adjusting the write intervals in the VMM was an effective approach to moderating the speed of background copy.

## 6. Discussion

***Dedicated v.s. shared NIC:*** We made a decision to use a dedicated NIC for the VMM to transfer OS images via network. However, sharing a NIC with the guest OS by using a device mediator is technically possible.

To share a NIC, we create a shadow version of ring buffers. The shadow ring buffers are maintained by the VMM and the pointer to the buffers are set to the physical NIC. The guest ring buffers are maintained by the device driver of the guest OS and their contents are copied to and from the shadow ring buffers by the VMM. To perform the copy on the update of buffers, the VMM virtualizes the registers of head and tail pointers to the ring buffers in the NIC. The VMM interleaves its own network requests with the requests from the guest OS into the shadow ring buffers to share the NIC. Most of housekeeping operations on the NIC are performed by the guest device driver and the VMM needs to perform minimal virtualization operations.

This approach is effective while the guest device driver is working. At the OS boot stage when the network device driver is not initialized yet and the shutdown stage when the driver is already terminated, the VMM must handle the NIC with its own device driver. There is some complication on the transition between the VMM device driver and device mediator. The VMM must temporary virtualize the NIC while the guest device driver is performing the initialization and termination because such operations will stop the NIC and the VMM cannot use the network unless virtualized. However, it is possible to handle this situation by partially virtualizing some of hardware registers. We have implemented device mediators for Intel PRO/1000 and Realtek RTL8169. Another possible approach to sharing a NIC is to exploit virtual functions of SR-IOV.

Nevertheless, we did not choose to use a shared NIC mainly because of the performance reason. For example, using a device mediator in the VMM may increase latency and jitter of the guest network performance. These types of net-

work degradation are undesirable for bare-metal cloud applications because the network tends to be a critical path for the performance of such applications. Moreover, guest network throughput could significantly decrease if applications heavily access the local storage because the guest OS and VMM scrambled for the network bandwidth of the shared NIC. In addition, using the combination of device mediators and device drivers in the VMM a little bit complicates the implementation, as described above. Since it is commonplace for modern server machines to have multiple NICs, we believe using a dedicated NIC is a reasonable design choice.

***OS transparency v.s. implementation cost:*** There may be negative opinions against providing OS transparency in exchange for paying implementation cost of device mediators.

Our emphasis here is that providing OS transparency is very important. If the number of OS types used in bare-metal clouds is small, it is reasonable for cloud providers to prepare pre-configured OS images and customers just use them. However, we believe bare-metal cloud providers should support a wide range OSs to maximize the potential benefits for customers. For example, customers may want to use research kernels, such as multikernels (e.g. Barrelfish [19]), microkernels (e.g. L4 [25]), or library OSs (e.g. Exokernel [26]), for the performance reason. Even if they use Linux or Windows, they may want to use various types, versions, and distributions of such OSs and heavily customize their configuration to take advantage of bare-metal clouds. Installing and maintaining customized drivers for various instances are daunting tasks for both providers and customers. Device mediators are, once implemented in the provider side, beneficial for all (many) customers by removing such tasks.

We also emphasize that the cost of implementing device mediators is not so high. Like device drivers for commodity OSs, device mediators also require device-specific knowledge and implementation cost. However, implementation of device mediators is simpler than that of device drivers because device mediators do not need to handle all I/Os but need to handle only key I/Os which are relevant to I/O redirection and multiplexing. For example, they can basically omit handling operations such as device initialization, power management, revision-specific trivial configuration and workarounds for errata.

In addition to the cost of implementing a single device mediator, the cost of supporting various devices may become a subject of discussion because a device mediator must be implemented for each type of devices. However, we believe this issue is not so critical because the variety of server hardware is limited compared to that of client and can be accommodated by software vendors. For example, VMWare ESXi maintains their own set of device drivers for various server hardware. This means that it is recognized as practical for a VMM vendor to support various server hardware.

Given formal device specifications, automatic synthesis of device mediators is theoretically possible and is a future work for covering various devices with minimal development cost. From a high-level viewpoint, device mediators have similar logic with device drivers in the sense that they perform device-specific operations responding to requests from guest OSs. Therefore, existing techniques for automatic device driver synthesis [42] can be applicable to the development of device mediators. Furthermore, device mediators are independent of OS-specific interfaces and are expected to be synthesized with more simplified method.

Overall, therefore, we believe the merit of providing OS transparency outperforms the cost of implementing device mediators.

## 7. Conclusion

Bare-metal clouds are an emerging and vital service, but the long startup time of instances has been a crucial limitation in providing agility and elasticity. We proposed BMcast, an OS deployment system with a special-purpose de-virtualizable VMM that supports OS-transparent quick startup of bare-metal instances. BMcast performs low-overhead streaming OS deployment transparently via copy-on-read and background copy to local storage from the server, and then disappears after completing the deployment. To achieve seamless de-virtualization of I/O devices, we introduced device mediators that perform polling-based device-interface-level I/O mediation (I/O interpretation, I/O redirection, and I/O multiplexing), allowing physical devices to be directly exposed to and shared with the guest OS.

We implemented a prototype VMM based on BitVisor that supports device mediators for IDE and AHCI devices, small network device drivers for NICs dedicated for the VMM, and a network-storage protocol that extends the AoE protocol. BMcast can deploy Windows and Linux without any modifications and startup a database instance 8.6 times faster than image copying. The database throughput on BMcast during streaming deployment was comparable to that on KVM with ELI. After de-virtualization, BMcast incurred zero overhead. Therefore, we confirmed that BMcast achieved quick startup of bare-metal instances, seamless de-virtualization, and eventual bare-metal performance.

## Acknowledgments

We would like to thank the anonymous reviewers for their helpful and constructive comments. We would also like to thank Hideki Eiraku, Tomohiro Kitamura, Katsuya Matsubara, and all others who were involved in the development of BitVisor. This work was partially supported by Adaptable & Seamless Technology Transfer Program through Target-driven R&D (A-STEP) from Japan Science and Technology Agency (JST), JSPS KAKENHI Grant Number 24500031, and Initiative on Promotion of Supercomputing for Young or Women Researchers, Supercomputing Division, Information Technology Center, The University of Tokyo.

## References


[1] ELVIS/ELI I/O acceleration code. http://lists.gnu.org/archive/html/qemu-devel/2013-09/msg04610.html.

[2] Facebook: Virtualisation does not scale. http://www.zdnet.com/facebook-virtualisation-does-not-scale-4010021998/.

[3] Containers at scale. https://speakerdeck.com/jbeda/containers-at-scale.

[4] Internap Network Services Corporation. http://www.internap.com/.

[5] Liquid Web Inc. http://www.liquidweb.com/.

[6] OSU Micro-Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[7] Openstack. http://www.openstack.org/.

[8] Peer 1 Hosting. http://www.peer1.com/.

[9] Rackspace, Inc. http://www.rackspace.com/.

[10] SoftLayer Technologies, Inc. http://www.softlayer.com/.

[11] Sysbench. https://launchpad.net/sysbench.

[12] Webair Internet Development, Inc. http://www.webair.com/.

[13] fio – Flexible IO Tester. http://git.kernel.dk/?p=fio.git.

[14] ioping. https://code.google.com/p/ioping.

[15] ATA over Ethernet Tools. http://aoetools.sourceforge.net/.

[16] M. Aron and P. Druschel. Soft Timers: Efficient Microsecond Software Timer Support for Network Processing. *ACM Transactions on Computer Systems (TOCS)*, 18(3):197–228, Aug. 2000.

[17] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, Oct. 2003.

[18] S. K. Barker and P. Shenoy. Empirical Evaluation of Latency-sensitive Application Performance in the Cloud. In *Proceedings of the 1st annual ACM SIGMM Conference on Multimedia Systems*, MMSys '10, pages 35–46, Feb. 2010.

[19] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, 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*, SOSP '09, pages 29–44, Oct. 2009.

[20] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 423–436, Oct. 2010.

[21] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 26–35, Mar. 2008.

[22] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The Collective: A Cache-Based System Management Architecture. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, pages 259–272, May 2005.

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud Computing*, SoCC '10, pages 143–154, June 2010.

[24] C. David, G.-E. Luis, and R. Sean. OS Streaming Deployment. In *Proceedings of the 29th IEEE International Performance Computing and Communications Conference (IPCCC)*, pages 169 –179, Dec. 2010.

[25] K. Elphinstone and G. Heiser. From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels? In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 133–150, Nov. 2013.

[26] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, Dec. 1995.

[27] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. ELI: Bare-metal Performance for I/O Virtualization. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 411–422, Mar. 2012.

[28] Internap, Inc. Cloud Landscape Report: Price & Performance Obstacles Surface for Fast, Big Data Applications, Jan 2014.

[29] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):931–945, June 2011.

[30] 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, June 2010.

[31] A. Kivity. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[32] T. Kooburat and M. Swift. The Best of Both Worlds with On-Demand Virtualization. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems*, HotOS XIII, May 2011.

[33] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, Apr. 2010.

[34] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable Virtual Machines Enabling General, Single-node, Online Maintenance. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 211–223, Oct. 2004.

[35] M. Mao and M. Humphrey. A Performance Study on the VM Startup Time in the Cloud. In *Proceedings of the IEEE 5th International Conference on Cloud Computing*, pages 423–430, June 2012.

[36] J. C. Mogul, J. Mudigonda, J. R. Santos, and Y. Turner. The NIC Is the Hypervisor: Bare-Metal Guests in IaaS Clouds. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS XIV, May 2013.

[37] V. Nae, R. Prodan, T. Fahringer, and A. Iosup. The Impact of Virtualization on the Performance of Massively Multiplayer Online Games. In *Proceedings of the 8th Annual Workshop on Network and Systems Support for Games*, NetGames '09, page 9, Nov. 2009.

[38] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing. In *Cloud Computing*, volume 34 of *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering*, pages 115–131. Springer Berlin Heidelberg, 2010.

[39] H. Raj and K. Schwan. High Performance and Scalable I/O Virtualization via Self-Virtualized Devices. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, HPDC '07, pages 179–188, June 2007.

[40] K. Razavi and T. Kielmann. Scalable Virtual Machine Deployment Using VM Image Caches. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, page 65, Nov. 2013.

[41] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 199–212, 2009.

[42] L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser. Automatic Device Driver Synthesis with Termite. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 73–86, Oct. 2009.

[43] B. C. S. Hopkins. AoE (ATA over Ethernet). February 2009.

[44] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, SOSP '07, pages 335–350, Oct. 2007.

[45] J. Shafer. I/O Virtualization Bottlenecks in Cloud Computing Today. In *Proceedings of the Second Workshop on I/O Virtualization*, WIOV'10, Mar. 2010.

[46] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. BitVisor: A Thin Hypervisor for Enforcing I/O Device Security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 121–130, Mar. 2009.

[47] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards Scalable Multiprocessor Virtual Machines. In *Proceedings of the 3rd Virtual Machine Research And Technology Symposium*, pages 43 – 56, May 2004.

[48] G. Wang and T. S. E. Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *Proceedings of the 29th IEEE International Conference on Computer Communications*, INFOCOM 2010, pages 1163–1171, Mar. 2010.

[49] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel. Concurrent Direct Network Access for Virtual Machine Monitors. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 306–317, Feb. 2007.

[50] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 203–216, Oct. 2011.