GiantVM: A Type-II Hypervisor Implementing Many-to-one Virtualization

Jin Zhang, Zhuocheng Ding, Yubin Chen, Xingguo Jia, Boshi Yu, Zhengwei Qi, Haibing Guan
Shanghai Jiao Tong University
{jzhang3002,tcbbd,binsschen,jiaxg1998,201608ybs,qizhwei,hbguan}@sjtu.edu.cn

Abstract

In recent years, since scale-up machines are not economical and may not be affordable for small businesses, scale-out has become the standard answer to data analysis, machine learning, and many other fields. However, these frameworks introduce complex programming models that put a burden on developers. Therefore, Single System Image (SSI), which means a cluster of machines that appears to be one single system, has been proposed to hide the complexity of distributed systems. Unfortunately, due to the mature ecosystem of current mainstream Operating Systems (OSes), it might be non-trivial and even unaffordable to modify the current OS to implement SSI. With the wide use of virtualization, we believe that it is appealing to support SSI at the hypervisor, without modifying guest OSes.

This paper presents GiantVM, an open-source distributed hypervisor that provides the many-to-one virtualization to aggregate resources from multiple physical machines, as well as providing a uniform hardware abstraction for guest OS. GiantVM combines the benefits of scale-up and scale-out solutions, which means unmodified applications are able to run with a huge amount of physical resources. Furthermore, GiantVM leverages distributed shared memory to achieve aggregation of memory. We also propose techniques to deal with the challenges of CPU and I/O virtualization in distributed environments. We have implemented GiantVM based on a state-of-the-art type-II hypervisor QEMU-KVM, and it can currently host conventional OSes such as Linux. Evaluations identify the performance bottleneck and show that GiantVM outperforms Spark by up to 3.4X with two text-processing programs.

CCS Concepts • **Software and its engineering → Virtual machines**; **Distributed memory**;

Keywords Resource aggregation, Virtualization, Single system image, Distributed shared memory

ACM Reference Format:
Jin Zhang, Zhuocheng Ding, Yubin Chen, Xingguo Jia, Boshi Yu, Zhengwei Qi, Haibing Guan. 2020. GiantVM: A Type-II Hypervisor Implementing Many-to-one Virtualization. In *16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20), March 17, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3381052.3381324

VEE '20, March 17, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7554-2/20/03…$15.00
https://doi.org/10.1145/3381052.3381324

1 Introduction

As the speed of information growth exceeds Moore's Law, we have seen a paradigm shift from scale-up to scale-out at the beginning of this century. To deal with the big data problem, data processing frameworks such as MapReduce [29] and Spark [65] are proposed and quickly become prevalent. Today, the conventional wisdom is that traditional scale-up solutions such as mainframes [20, 60] have high cost-performance ratios and cannot support the large-scale businesses of web companies.

However, scale-out data processing frameworks introduce complex programming models that put a burden on developers. There is a dilemma between the configuration difficulties, programming model complexities [28], runtime overheads of scale-out distributed frameworks, and the high price of scale-up machines for those tasks whose data set cannot reside in a single normal machine.

To overcome this dilemma, the Single System Image (SSI), which runs a single Operating System (OS) instance on a cluster of machines [27], has been proposed to hide complex distributed programming models. However, the traditional SSI usually is a distributed OS or cluster OS [48, 52, 56]. These approaches usually involve significant engineering complexity to provide the compatibility of the existing software ecosystem (e.g., POSIX). Further, many heterogeneous device drivers are not open-source, thus are impractical to port to a dedicated OS.

In parallel with this rising of big data, virtualization has become another important trend since the 2000s. Through the powerful abstraction of virtualization, cloud providers are able to provide a virtual machine to achieve simpler management and better resource utilization, without modifying the guest OS.

To exploit both SSI and virtualization, previous attempts have been made to combine them based on Distributed

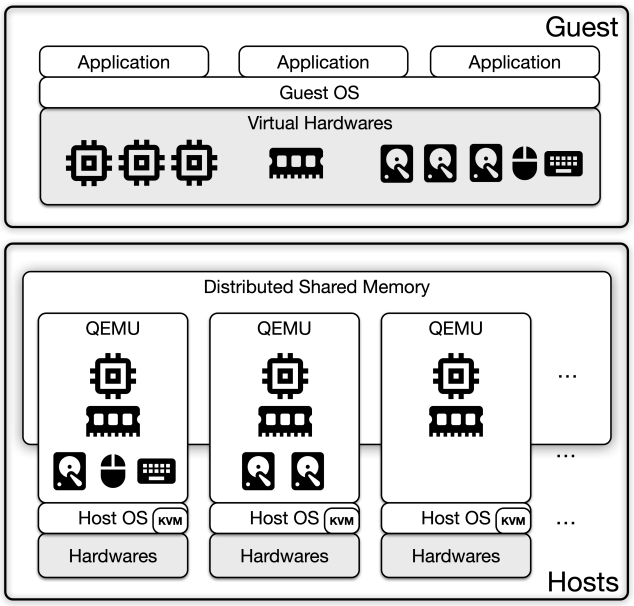


Figure 1. The architecture of GiantVM

Shared Memory (DSM) [3, 7, 27]. However, these type-I virtualization approaches require special hardware with a dedicated hypervisor. For example, vNUMA is a type-I distributed hypervisor on Itanium leveraged by pre-virtualization [39] while TidalScale [7] needs a dedicated HyperKernel. Similar to the distributed OS, these customized type-I hypervisors are difficult to deploy and maintain on the cloud environment.

In contrast to the type-I virtualization, QEMU-KVM is a dominant type-II hypervisor in the cloud environment. Why not build a general purpose SSI based on QEMU-KVM? A Type-II hypervisor can reuse utilities provided by the host OS, especially the heterogeneous device drivers. It is difficult or even impractical to port such software to a type-I hypervisor. In this paper, we present a QEMU-KVM based distributed type-II hypervisor, GiantVM, which builds an easy-to-use SSI platform for aggregating computing, memory, storage, and I/O resources on which the general unmodified guest OS like Linux can run.

This paper presents challenges and implementation details of I/O, CPU, and memory virtualization under a distributed environment. For the I/O virtualization, a master node is designated and I/O requests are multiplexed by it. For the CPU virtualization, it works independently on each node at most time, except for the inter-core communication between two nodes which we should focus on. The memory virtualization is a DSM based on Ivy [40]. We illustrate the details of integrating DSM into a hypervisor. We also trigger Remote Direct Memory Access (RDMA) to accelerate the network transmission.

In the evaluation part, we analyze several micro-benchmarks using Sysbench [6], and identify the performance bottlenecks. Our experiments also show that GiantVM can outperform Spark by 3.4X for specific applications.

In summary, our key contributions are outlined as follows:

- We propose GiantVM, the first distributed type-II hypervisor, providing the many-to-one virtualization to aggregate resources from multiple physical machines.
- We implement an open-source prototype based on QEMU-KVM and integrate distributed shared memory into it, including *IPI forwarding*, *interrupt forwarding*, *I/O forwarding*, and DSM-related optimization techniques, which overcome the difficulties of sharing CPU, memory, and I/O devices across machines.
- We evaluate our distributed hypervisor implementation using a standard Linux as the guest OS and run several benchmarks including Sysbench [6], stress-ng [5], and Spark [65] to identify the performance bottlenecks and propose the solution.

The rest of this paper is organized as follows. Section 2 describes the background of GiantVM. In Section 3, we illustrate the design principles of GiantVM. And in Section 4, we elaborate on the implementation of GiantVM. The evaluation of GiantVM lays in Section 5. We discuss our work in Section 6. In Section 7, we present the related work. Finally, we conclude our work in Section 8.

2 Background

2.1 System Virtualization

System virtualization refers to an abstraction layer that separates the physical hardware from OSes to gain more flexible resource utilization, which enables multiple virtual machines to run on a single physical machine. System virtualization is traditionally implemented purely in software, but hardware-assisted virtualization approaches have been prevalent nowadays. On x86 platforms, Intel [58] and AMD [11] both have their hardware virtualization solutions. We mainly focus on Intel's technologies in this paper.

The software providing the abstraction is called a Virtual Machine Monitor (VMM) or hypervisor. It can be classified into two types, type-I and type-II. A type-I hypervisor runs on a bare metal directly while a type-II hypervisor relies on a host OS. The type-I hypervisor has less overhead, but it cannot reuse the tools provided by the host OS like the type-II hypervisor.

CPU Virtualization Intel's VT-x extension introduces a new mode called VMX and a series of VMX instructions. VMX mode is further divided into *non-root* mode and *root* mode, in which the guest OS and hypervisor reside respectively. The transition from non-root mode to root mode is referred to as VMExit. Under non-root mode, there are some privileged instructions which will cause a VMExit when

executed, and the hypervisor only needs to emulate those instructions instead of emulating each instruction.

Memory Virtualization Traditionally, memory is virtualized through shadow page tables. Intel introduces Extended Page Table (EPT) which provides secondary mappings from Guest Physical Address (GPA) to Host Physical Address (HPA). Given a Guest Virtual Address (GVA), it is first translated to GPA using the conventional page table and then translated to HPA using EPT. If the translation through EPT fails (e.g., due to an invalid or read-only entry), a special VMExit called *EPT violation* will be triggered. We also use *page fault* to refer to EPT violation in this paper.

I/O Virtualization Traditionally, I/O devices are virtualized through the trap-and-emulate approach. Intel's VT-d [8] introduces an IOMMU and makes device passthrough efficient. Specifically, devices generally use Direct Memory Access (DMA) to transfer data to and from memory. With IOMMU, the address used by DMAs becomes virtual address and is translated by IOMMU to physical address. With the help of IOMMU, guest OSes can directly issue DMAs using GPA, and the GPA will be translated to HPA by IOMMU automatically.

QEMU-KVM QEMU-KVM is a popular open-source hypervisor. KVM [37] is a Linux kernel module that provides support for hardware-assisted virtualization. QEMU [19] is a hypervisor on its own, running in the userspace. It can use KVM as its execution backend, delegating the virtualization of CPU and memory to KVM, and it handles the emulation of I/O devices.

2.2 Distributed Shared Memory

DSM [49] provides a continuous virtual memory address space crossing multiple physical machines for applications. In a DSM system, each CPU can use virtual addresses to access memory, without being aware of the actual data location. The research on DSM faces its first peak in the 1990s. Today, we believe the evolving high-performance networks will provide new opportunities to DSM.

2.3 RDMA

RDMA is a recent network technology, and RDMA capable networks such as Infiniband usually provide low latency and high throughput compared to the traditional Ethernet and TCP/IP combination [14, 43]. The average latency of TCP is 9 times higher than RDMA [66]. There are two groups of RDMA API, *one-sided* and *two-sided*. One-sided RDMA allows local CPUs to operate remote memory directly, without the involvement of remote CPUs. Two-sided RDMA is similar to traditional channel-style APIs such as socket. One-sided RDMA usually has lower latency than two-sided RDMA, but the opposite conclusion can be drawn when one transaction can be completed by either multiple one-sided operations or single two-sided operation [35, 55].

3 Design

3.1 Principles

Transparency from applications We aim to implement the SSI at the hypervisor layer, which effectively provides a unified virtualized Instruction Set Architecture (ISA) interface. The design enables all OSes supporting the x86 platforms, as well as the applications built for the OSes.

A type-II hypervisor built on the x86 platforms First, although type-I hypervisor has less overhead, the support for hardware drivers including RDMA, GPU, and any other heterogeneous devices in the host OS of a type-II hypervisor is vital to GiantVM. Besides, the convenience of deployment of type-II hypervisor plays a key role in modern cloud computation infrastructure. Second, x86 dominates compute-intensive workstation and cloud computing segments. As a result, most software of these areas is designed for x86 platforms. Of course, it is feasible to apply our approach to other architectures.

A dedicated hypervisor rather than kernel The type-II hypervisor is usually light-weight for it can reuse tools provided by the host OS. We decide to implement all functions of GiantVM in the hypervisor. The alternative that implements some functions in the host OS is complex and error-prone. It also breaks the principle that the host OS usually treats the type-II hypervisor and the VMs as common processes.

3.2 Overview

The architecture of GiantVM is shown in Figure 1. From the view of applications, the standard ISA interface is provided. The underlying infrastructure consists of a cluster of physical machines. There is a *hypervisor instance* running on each machine. To make multiple hypervisor instances cooperate to provide a single consistent view of virtualized hardware for the guest, we distinguish local and remote resources. That is, each hypervisor instance on different physical machines creates a complete virtual machine image using the same parameters, but each with some hardware resources marked as remote and delegated to other hypervisor instances. From the view of a hypervisor instance, it just starts an ordinary VM, except that some resources are managed by remote hypervisor instances. An exception to this design is the memory, which is shared by all instances through the DSM.

3.3 Distributed vCPU

Firstly, we make hypervisor instances aware of the distinction between local and remote CPUs through command line parameters. For local vCPUs, they are treated as usual, with the exception that some instructions involving shared (i.e.,

memory) or remote resources are treated specially. For remote vCPUs, their threads and data structures are initialized and the threads are yielded until the termination of the VM. The reason why we initialize remote vCPUs is to provide dummy APICs (see below).

Once we have vCPUs running, the next thing to consider is Inter Processor Interrupt (IPI), which only involves the interaction of vCPUs. On x86 architectures, there is an Advanced Programmable Interrupt Controller (APIC) in each core, which is responsible for accepting and delivering interrupts.

In short, we propose an *IPI forwarding* mechanism. Its overall strategy is straightforward. We add a check at places that will issue an IPI. If the IPI is sent to local vCPU, then we just process it as usual; otherwise, we send it to remote hypervisor instance and inject it into the APIC of remote vCPU.

3.4 Distributed Memory

Memory virtualization is the most critical part of GiantVM. Since the memory model of x86 platforms is x86-TSO [53], we must implement DSM with consistency that is at least not weaker than x86-TSO. x86-TSO is a fairly strong consistency model and leaves the underlying DSM limited choice. We discuss the possibility of relaxing memory model in Section 6.1. Based on the description above, we achieved it through a sequential consistency DSM based on the protocol of Ivy [40].

There are three states in the DSM, namely *Modified*, *Shared*, and *Invalid*. Each page can be in one of the three states. If it is in Modified state, it has the exclusive ownership of this page and can do reads and writes at will. If it is in Shared state, it shares the page with other nodes and is write protected. If it is in Invalid state, the page is marked not present in page table and must ask other nodes for the latest copy of this page before any read or write can be done. Similar to the "improved" version of the protocol described in Ivy, we have the concept of owner for a page, which is the node in Modified state, or the node maintaining a *copyset* for Shared copies of the page. We designate managers to track the owner of pages. Every node is the manager which is responsible for a portion of guest memory space, and this matches nicely with Non-uniform Memory Access (NUMA).

When a read occurs on an Invalid page, or a write occurs on a Shared or Invalid page, a page fault is triggered, and the execution is transferred to the hypervisor instance. Then a request to the manager of this page is issued, shown as step ① - ③ in Figure 2. The manager then forwards the request to the owner of this page, as shown in step ④ of Figure 2 (b), or it processes the request immediately if it is already the owner, as shown in Figure 2 (a). For read requests, the requesting node is added to the copyset of the owner and is sent to the latest copy of this page. For write requests, the old owner invalidates all the holders of this page, and then transfers the ownership and the latest copy of this page to the requesting node. Those are illustrated as step ④ - ⑥ / ⑤ - ⑧ in Figure 2 (a) / (b) respectively. Finally, the hypervisor instance sets the page table so that the guest will no longer page fault, and then transfers the control back to the guest, as shown in step ⑦ - ⑧ / ⑨ - ⑩ of Figure 2 (a) / (b).

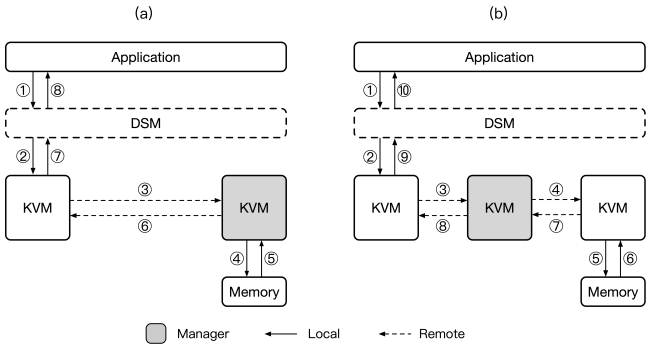


Figure 2. Typical operations of DSM

3.5 Distributed I/O

GiantVM should correctly emulate two behaviors about I/O devices, (1) interrupts from remote I/O devices, (2) I/O accesses to these devices. There are some dedicated methods for specific devices to achieve them. For example, a hypervisor instance can take initiative to have all I/O requests and interrupts of a disk device processed normally. And the global consistent disk view is maintained by the host OS via NFS or iSCSI. However, such methods are lack of generality and we desire a universal approach. Another point we concern about is that the natural property of the centralization of an I/O device. For example, a device driver has to multiplex it to support multiple clients. It leads to the necessity of a master node where the physical devices exist. Currently, we assign all devices to one master node. But it is possible to assign different devices to different nodes for the sake of load-balancing.

Our idea is straightforward. First, whenever an interrupt is generated by the device, it is captured by the hypervisor instance. GiantVM intercepts it and forwards it to the target vCPU which may be a remote one. Second, when the vCPU tries to issue an I/O request, it is captured by the hypervisor instance. In the same way, GiantVM intercepts it and sends the request to the master node accordingly.

4 Implementation

Our open-source prototype implementation of GiantVM[1] is based on QEMU 2.8.1.1 and KVM (Linux kernel 4.9.76) on the x86 platform with around 7K lines of code. The hypervisor instance is the modified QEMU-KVM hypervisor running on each node.

[1] https://github.com/GiantVM

To construct a distributed hypervisor, we need to virtualize CPU, memory, and I/O in a distributed way. For CPU and I/O, we add Interrupt and I/O Forwarding mechanisms to QEMU, which are described in Section 4.1 and 4.3. For memory, we implement a DSM module in KVM and adapt QEMU for it, which is described in Section 4.2. Besides, we also implement a proper time synchronization to make time-related functions like `sleep` work correctly. All the mechanisms we implement to support GiantVM is shown in Figure 4.

We support both traditional TCP/IP and RDMA as our network backend, to leverage the increasingly common high-speed and low-latency interconnections. Due to the inherent request-reply pattern of DSM and performance considerations (see Section 2.3), only two-sided RDMA is used at this time.

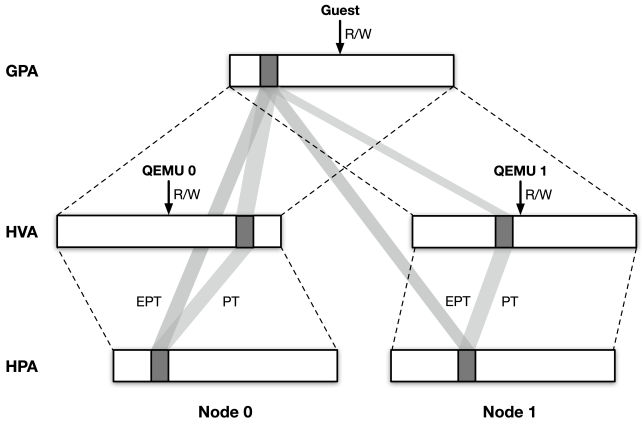


Figure 3. Memory mapping in QEMU-KVM

4.1 Distributed vCPU

The implementation based on the design is conventional. A local vCPU is implemented as a normal thread of the QEMU process and a remote vCPU just sleeps and waits for the exit of the VM. However, there is a technical detail of APIC that makes things more complicated. On x86 platforms, all IPIs (and actually all interrupts) are implemented as broadcast messages under the hood. Upon reception of an IPI message, the CPU uses three registers of APIC, namely APIC ID Register, Logical Destination Register (LDR), and Destination Format Register (DFR), along with the destination field of IPI message, to determine whether it is the destination of this IPI. In a distributed environment, as we cannot check the APIC of remote vCPUs before the IPI is sent to them, we initialize the remote vCPUs and their APICs locally. These APICs are *dummy APIC*s, since their only purpose is to provide the three registers necessary for the decision of IPI destination. Whenever a local vCPU modifies one of the three registers, it will do a broadcast to make all dummy APICs in corresponding remote vCPUs up to date. In this way, we can always know the target of an IPI and forward IPIs to remote vCPUs accordingly.

4.2 Distributed Memory

We integrate the DSM functionality into KVM because it can take advantage of the memory mapping information maintained by KVM and modify the EPT easily when a DSM page status is changed. And the interaction with the memory management module of host OS is elegantly decoupled with the `MMU Notifier`. However, the combination of hardware virtualization and DSM imposes challenges on the implementation and we propose solutions to them and describe how to tackle them as follows.

4.2.1 Mapping Management

In the presence of hardware virtualization, the mapping of memory is not as simple as in traditional DSMs. We can see this in Figure 3. First, QEMU allocates the memory to be used by the guest in its address space, i.e., Host Virtual Address (HVA) space. Then it is registered to KVM, and KVM maintains the mapping between GPA and HVA. When guest accesses a memory location, it first uses page table to translate the GVA to GPA, then uses EPT to translate the GPA to HPA.

There are some important facts in this scheme: (1) The page table we use to write protect or invalidate a page is EPT, which controls the GPA-to-HPA mapping, since it would be too intrusive to manipulate host page table directly. (2) The memory is allocated in HVA space, and for some devices, there are two MMIO memory regions in GPA space corresponding to the same memory region in HVA space, i.e., the HVA-to-GPA correspondence is a one-to-multiple mapping. (3) Since the only shared memory space between two nodes is the GPA space, we can only use GPA in our DSM messages.

Taking these points into consideration, we come up with the following design. The state and copyset are bound to the pages in HVA space instead of pages in GPA space, to avoid inconsistent states for two guest pages mapping to the same host page. When a host page is invalidated, the entries of all corresponding guest pages are cleared in EPT. Similarly, when a host page enters Shared state, all the corresponding guest pages are write protected. Further, since we transfer GPA in DSM messages, we need to translate the GPA back to HVA so that we can find the host page and its state. KVM only tracks **guest** pages and the corresponding data structures may be manipulated by other threads. For example, the swapping events on the guest memory issued by the host OS will drop the EPT entries. So additional data structures are necessary on the grounds of the tracking of **host** page status and the reverse mapping from host page to guest pages.

Additionally, the Dirty/Access bit support of EPT is disabled in the implementation. When this feature is enabled, any access to the guest page table, including page walks performed by hardware, is considered by EPT as a write when the GPA to HPA translation happens. The resultant

write faults will cause disastrous page thrashing and ruin the performance, so we intentionally disable this feature. However, the function of Dirty/Access bit can be emulated by the traditional software approach, i.e., setting read/write protect to target pages. In addition, the first generation of EPT does not have this Dirty/Access bit. This feature was added later to facilitate the virtual machine's dynamic migration. So for compatibility, it is an optional feature that can be disabled.

4.2.2 Memory Accesses Bypassing EPT

With the help of permission bits in EPT, we can trap all the accesses from the guest to Shared and Invalid pages. However, as the memory of the guest is actually allocated in HVA space, QEMU and KVM can manipulate it directly without involving the translation provided by EPT. Considering the DMA procedure of a disk I/O request, if the master QEMU writes data to an HVA space while the space is marked as Invalid in the EPT, other nodes can never read the correct data. Other examples of bypassing EPT include para-virtualized kvmclock and instruction emulators in KVM.

To solve this problem, we add two operations, *pin* and *unpin*, to the DSM. All direct manipulations of the HVA space are protected by these two operations (like lock and unlock) to ensure the integrity of the DSM protocol. When we *pin* a page, we first invoke a page fault to transfer it into the desired state, i.e., Shared or Modified, depending on whether the following manipulations are read-only or write. Note that there are fast-paths that the current state has already been the desired one. And then block all the incoming requests on this page, until an *unpin* on the same page is performed. Such blocking is rare since the common cases of memory accesses bypassing EPT are DMA. There is usually only one vCPU issuing DMA and subsequently accessing the corresponding pages. For accesses within KVM, we just use these internal operations to protect the accessed memory. For accesses in QEMU, we expose the operations through an ioctl called `MEMPIN`, which can do both *pin* and *unpin* on a host memory region. Then we can protect those accesses using the `MEMPIN` ioctl.

4.3 Distributed I/O

In a distributed environment, I/O devices can be scattered over different machines. For a remote I/O device to work properly, two things must be handled, that is, (1) interrupts from remote I/O devices, (2) I/O accesses to those devices.

4.3.1 Interrupt Forwarding

To handle interrupts sent from I/O devices to vCPUs on remote QEMUs, we use an *interrupt forwarding* approach similar to the one proposed in Section 3.3.

On x86 platforms, there are two ways for devices to send an interrupt, that is, through IOAPIC or Message Signaled Interrupt (MSI). MSI is the modern way and it refers to a method of generating interrupts by doing particular PCI

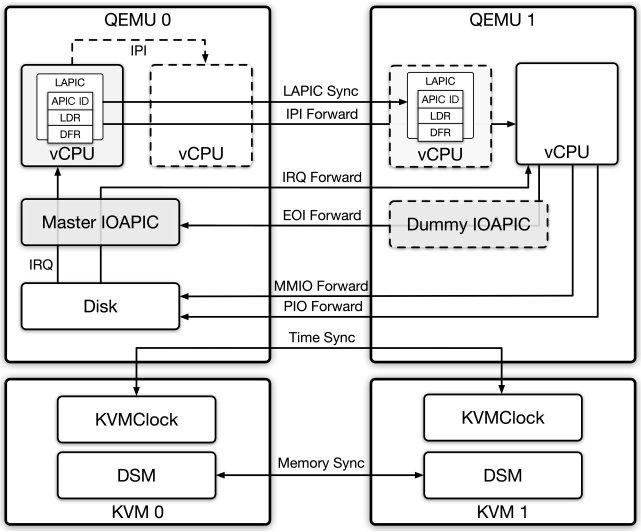


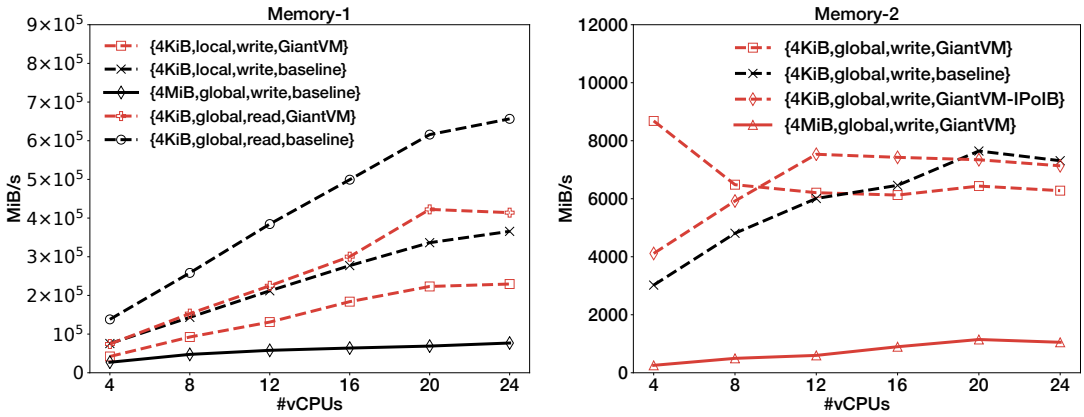
Figure 4. The inside of GiantVM

memory transactions, while IOAPIC is the old school interrupt handling chip originated in the 1990s.

Due to the difference between those two methods, we have different strategies for IOAPIC based interrupts and MSIs. One important difference between IOAPIC and MSI is that multiple legacy devices share one IOAPIC for interrupt configuration, interrupt queueing, and interrupt generation, while each modern PCI/PCIe device configures and generates interrupts on its own. Consequently, legacy devices need to inform the IOAPIC through sideband interrupt signals first so that the IOAPIC will generate interrupts for them.

The interrupt forwarding process for IOAPIC has been shown in Figure 4. As there should only be one IOAPIC per system, we designate one of the QEMUs as master (shown as QEMU 0) and its IOAPIC is the master IOAPIC. Other QEMUs are slaves, and their IOAPICs are dummy IOAPICs. Since legacy devices need to inform the master IOAPIC through interrupt signals, we initialize dummy IOAPICs to collect those signals and forward them to the master IOAPIC. Reads and writes to registers of IOAPIC are all routed to the master IOAPIC, so it contains all the necessary configurations and is capable of generating interrupts once the forwarded signals arrive. If the destination of a generated interrupt is a remote vCPU, which can be determined through the help of dummy APICs, it is forwarded as described in Section 4.1. Also, IOAPIC requires an End of Interrupt (EOI) message from CPUs for certain interrupts, i.e., level-triggered interrupts, while MSI does not. So we also need to send the EOI back to the master IOAPIC from the target vCPU when necessary.

For MSIs, things are much simpler. If accesses to the configuration space of a device are properly forwarded as described in Section 4.3.2, the MSI of this device will be configured



(a) Sysbench Memory-1 (b) Sysbench Memory-2

Figure 5. Sysbench memory results

correctly. Then all we need is to forward the interrupt to remote vCPUs when an MSI is generated by this device.

4.3.2 I/O Forwarding

Generally, I/O devices are accessed and programmed through memory mapped I/O (MMIO), while on x86 platforms there is an additional I/O address space that can only be accessed through special port I/O (PIO) instructions. Once the vCPU performs a PIO or MMIO, it will trap to KVM and then exit to QEMU if the PIO/MMIO cannot be handled in KVM (which is mostly the case, and we do not support devices emulated in KVM currently). When KVM exits due to a PIO/MMIO, we can add a check, and if the PIO/MMIO is to a remote device, we forward it to the remote node. After the access is processed on the remote side, a reply is sent back, and then we can finish this I/O access and return to guest mode again.

For PCI/PCIe devices, the situation is more complicated. Each device has its own configuration space which is accessed indirectly through the port `0xCF8` (address), and `0xCFC` (data), and we need to distinguish accesses to the configuration space of local and remote devices. Further, the MMIO memory region is dynamically configured through this configuration space, so we need to broadcast this configuration so that we can distinguish MMIOs to remote devices and illegal MMIOs.

Currently, our prototype has not implemented this design yet because it requires significant engineering efforts. Instead, we simply put all devices under the master QEMU (i.e., QEMU 0), and direct all PIO/MMIO accesses to it.

4.4 Time Synchronization

The virtualization of time has always been considered a hard problem [21]. It can be briefly summarized as *host_time* + *offset*, where *host_time* is backed by physical time-keeping services and *offset* is set by the hypervisor. Without the proper synchronization of *offset*, time-related functions like `gettimeofday` and `sleep` will misbehave consequently.

To make sure different nodes are in sync, we need to take two things into consideration: Time Stamp Counter (TSC) and kvmclock.

TSC is a register on x86 architectures, which increases at a constant rate continuously and thus can be used as a clock source providing elapsed time. VT-x provides two features called TSC offsetting and TSC scaling to adjust the value of TSC read by a guest vCPU, which gives:

$$guest\_tsc = host\_tsc * scale\_ratio + offset$$

To make sure TSC values seen by vCPUs are in sync, we designate the first started QEMU as the master, from which others will query the rate and value of TSC, derive the appropriate *scale_ratio* and *offset*, and then set them through KVM.

Kvmclock is a para-virtualized clock source maintained by KVM running at a fixed rate of 1GHz and is used as the primary clock source by guest OSes if enabled. The following formula gives the read value of kvmclock at time t:

$$kvmclock(t) = host\_boot\_time(t_0) + offset + (guest\_tsc(t) - guest\_tsc(t_0))/guest\_tsc\_rate$$

In which $host\_boot\_time(t_0)$ (time elapsed since boot on host) and $guest\_tsc(t_0)$ are updated periodically by KVM, $guest\_tsc(t)$ is read directly by guest vCPU at time t, and *offset* is a user-provided adjustment. To be precise, one cannot set *offset* directly, and can only get and set the value of kvmclock through KVM.

As the $host\_boot\_time(t)$ on different nodes differs from each other, we must provide an appropriate *offset* to keep the kvmclock read value on different vCPUs in sync. Fortunately, QEMU sets the kvmclock upon startup to make it start from 0. This process is modified to let slave QEMUs query the master

QEMU for its kvmclock value and apply it to themselves upon startup. Therefore, all vCPUs will see the same kvmclock value.

In both the TSC and kvmclock case, we also take round-trip time (RTT) of the query into consideration, and the time we used for slaves is actually:

$$slave\_time = master\_time + RTT/2$$

5 Evaluation

5.1 Experimental Setup

To examine the performance of GiantVM, we first use microbenchmarks to measure the performance of DSM and Forwarding mechanisms. And we analyze the performance results with several further experiments. Second, we choose applications which are CPU-intensive and I/O-intensive respectively. We also have a comparison with Spark on Linux.

All experiments are conducted on a cluster and a control group with one standalone machine on which baseline results of single-machine programs are measured. Their configurations are shown in Table 1. We disable hyper-threading on all machines.

Table 1. Experimental setup

Cluster	
Number of Machines	4
CPU	8-core Intel Xeon E5-2620 v4
DRAM	128GB
Ethernet NIC	Broadcom NetXtreme BCM5720 Gigabit Ethernet
RDMA HCA	ConnectX-3 MCX354A-FCBT 56Gbps InfiniBand
OFED Version	Mellanox OFED v2.2-1
Disk	SEAGATE ST9300605SS
OS	Ubuntu 16.04
Control Group	
CPU	24-core Intel Xeon E5-2650 v4
DRAM	64GB
OS	Ubuntu 16.04
Guest Configuration	
DRAM	64GB
OS	Ubuntu 16.04

We choose `sysbench` [6] as our microbenchmark, measuring the performance of memory, mutex, and I/O sub-systems and identifying the performance bottleneck. In the application part, we use CPU methods of `stress-ng` [5] to evaluate CPU-intensive applications. Finally, we compare `Word Count` and `Inverted Index` from scratch with jobs running on Spark. We use the `monospaced` fonts when describing an experiment and normal fonts when describing the specific software.

The baseline is the result of applications running on the vanilla QEMU-KVM hypervisor. All applications are allocated with sufficient memory, hence there is no swapping. No vCPUs are overcommitted. All experiment results are the average of five runs. We use (m nodes $*$ n vCPUs) to denote the configuration of GiantVM with m nodes, n vCPUs in each machine.

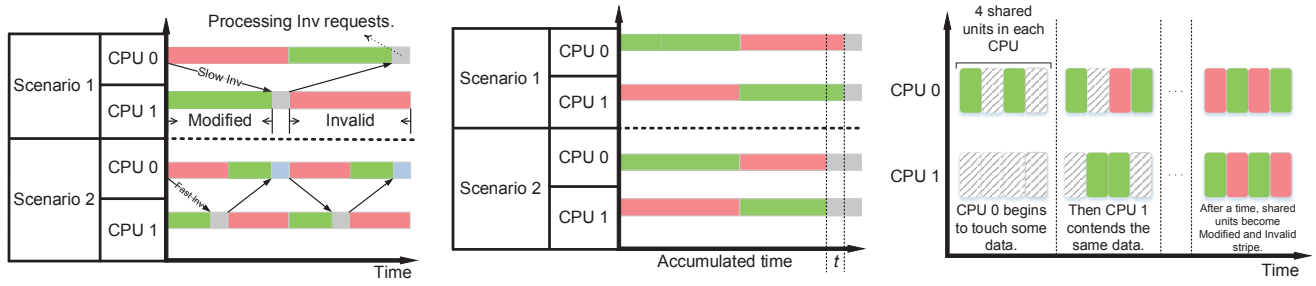
5.2 Sysbench

We first use `sysbench` to investigate the performance of GiantVM. The experiment configuration is (4 nodes $* N/4$ vCPUs) where N is the total number of vCPUs in each test.

Memory We use $\{size, scope, oper, mode\}$ to describe the memory test configuration. It is read as doing *oper* (read or write) in *scope* (local or global) memory on the *mode* (GiantVM or baseline). Local memory means each thread accesses its own page-aligned memory space while global memory means all threads access global shared memory. The size of memory accessed in each turn is *size* (4KiB or 4MiB). The result is shown in Figure 5. Note the similarity between MESI protocol used in x86 cache system and MSI protocol. When doing read-only operations or writes on local *shared units* (cachelines in cache system or pages in DSM system), both protocols allow all the processors to make progress at the full speed. Thus the results are given as optimized and the speed of access is *de facto* the speed of L1 cache in our machines. The reason for the performance degradation of GiantVM is that when a timer interrupt is fired, GiantVM needs to touch more data structures. This leads to more cacheline eviction comparing to the baseline.

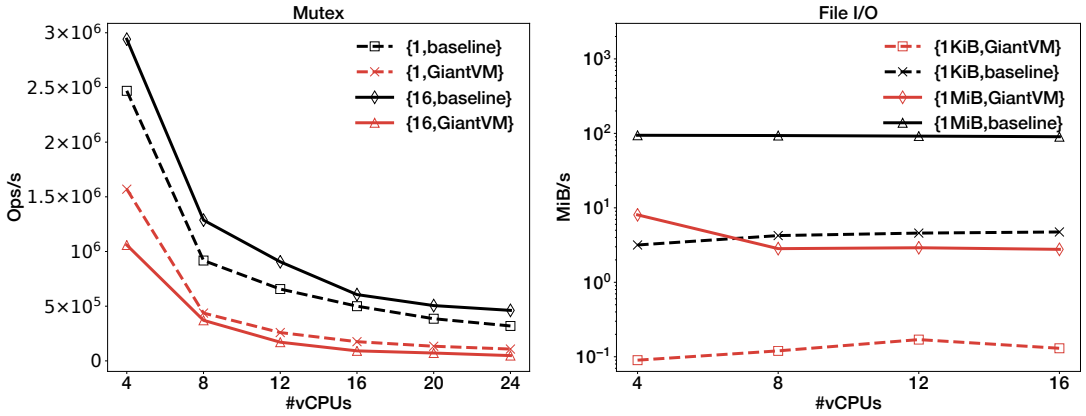
When we test $\{4K, global, write, *\}$, a dramatic phenomenon arises. DSM has a miss penalty of ~$15us$ while cache system is only ~100 cycles. However, GiantVM outperforms the baseline surprisingly. A further experiment is given by replacing RDMA network with IPoIB network (Ethernet is too slow to bootstrap Linux for the failures of timeout-based services). We observe an interesting "*slower is faster*" phenomenon. There exists some data that IPoIB > RDMA > baseline (DRAM & SRAM). To explain it, we illustrate two facts about the DSM system and cache system. First, note that when writing on the same shared units, only one writer can make progress (M(E)SI is a so-called single-writer protocol). Other vCPUs desiring write have to send an `Invalid` message first. If the message is sent too fast (like the cache system), it leaves little time for the current writer doing enough work. In an extreme scenario, it leads to the livelock. On the contrary, a slow message enables the current writer to complete more operations. In Figure 6 (a), Scenario 1 and 2 display the difference. Figure 6 (b) shows there is a difference t between a system with slow messages and a fast one. Second, if the working set spans across multiple shared units like Figure 6 (c), a CPU soon reaches an invalid shared unit even if a remote `Invalid` message has not arrived yet. The cache system with a 64B cacheline is the victim of the fact. It is possible that an optimization that delays a DSM request for a while can be conducted when such pattern is detected.

The phenomenon of "slower is faster" disappears when we test $\{4MiB, global, write, *\}$. The DSM now suffers from a large working set while it is sparse[2]enough for the cache system and there is little cacheline thrashing. At this time, a



(a) Thrashing of one shared unit (b) Accumulated time profile (c) Contention on multiple shared units

Figure 6. State transmissions of heavily-contented shared units. The red boxes mean the shared units of the CPU are Invalid states, while the green boxes mean the current states are Modified (the time when the application can make progress). If a CPU wants to write an Invalid shared unit, it must issue an Invalid request (represented as arrows in (a)) to the currently running CPU, leading to a state switch and overhead to deal with the Invalid request (gray boxes). (b) is depicted by accumulating time slices of different states, showing a downtime difference t. Shadowed boxes in (c) refer to unrelated data.



(a) Sysbench Mutex (b) Sysbench File I/O

Figure 7. Sysbench mutex and file I/O results

huge gap of accessing speed between L3 cache (or DRAM) and remote memory is shown.

We call such workload *throughput-oriented*, which means the writer can make progress without blocking. We discuss *latency-oriented* workload in the next section.

Mutex We use $\{nlocks, mode\}$ to describe the mutex test configuration. It is read as randomly picking a lock from a mutex array with *nlocks* locks, locking and unlocking it in each turn on the *mode* (GiantVM or baseline). The result is shown in Figure 7 (a). $\{1, *\}$ is similar to $\{4k, global, write, *\}$ but a key difference is that the former is latency-oriented. Unlike the throughput-oriented workload, CPUs are blocked in the blank areas of Figure 6. A CPU desiring locking must wait for another CPU unlocking, i.e., an incoming `Invalid` message and the subsequent read fault. Because of the latency difference of sending `Invalid` message, cache system has a significant advantage (a small blocked area) in this scenario.

$\{16, *\}$ shows the *false sharing* problem. Although at most time threads access the different locks, the locks are allocated in one page. Thus these accesses are dependent. The cache system has the same problem too. And it puts a burden on the programmers, asking them to allocate cacheline-alignment data structures (such as mutex). Applications running on the GiantVM can be tuned as changing the definition of cacheline size to 4KiB.

File I/O We use $\{size, mode\}$ to describe the file I/O test configuration. It is read as sequentially reading/writing a *size* (1KiB or 1MiB) file on the *mode* (GiantVM or baseline).

[2]Although we set the default sequential operations, there are no explicit barriers between threads. And the access is approximately random. Further, if the working set is large enough, access of each CPU can be treated as local and thus there is little thrashing.

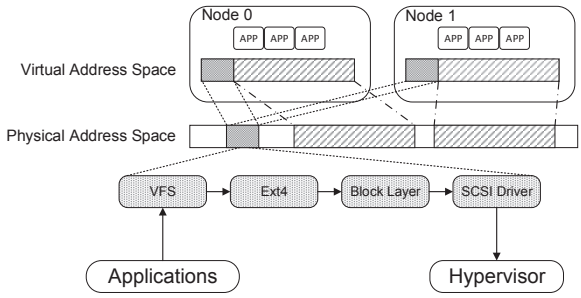


Figure 8. An overview of Linux storage stack. Although most memory space is separated by processes, a shared area of the kernel space causes catastrophic page contention. Specifically, most contention pages belong to the heap area of storage stack for an I/O intensive workload.

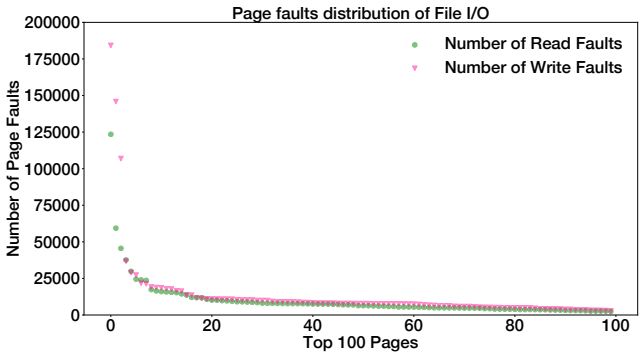


Figure 9. Proportion of page faults for top 10 frequently page-faulting pages

The result is shown in Figure 7 (b), which is a catastrophic consequence. We begin to analyze the problem with perf [2]. Table 2 denotes the DSM component occupies the most CPU time and is the bottleneck. From a high level view, in order to write bytes to the disk, one has to walk through a plethora of obstacles shown in Figure 8. The whole storage stack above the hypervisor requires the involvement of the DSM component. The storage stack fully considers the underlying cache system (i.e., allocation of cacheline-alignment data structure) but not for GiantVM. Several locks and shared data are allocated in one page, which causes severe false sharing and page thrashing. And such pages exist in every layer of the protocol stack. Another proof is the top 100 frequently page-faulting pages shown in Figure 9. The proportion of page faults happening on pages whose total memory size is less than 0.5MB is nearly 70%.

Table 2. The `perf` results

Components	Description	Samples
Router	Interrupt & I/O router	1.06%
QEMU	QEMU except Router	0.97%
GuestOS	Guest OS (non-root mode)	35.44%
DSM	DSM page fault handler	43.75%
Others	Kernel part except DSM	18.89%

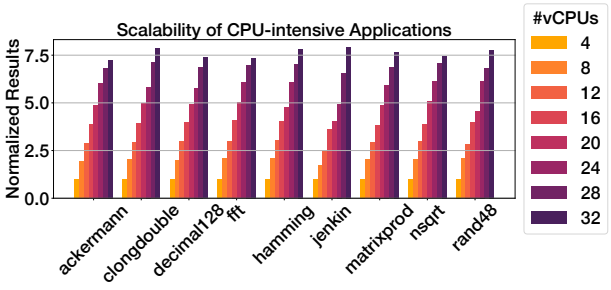


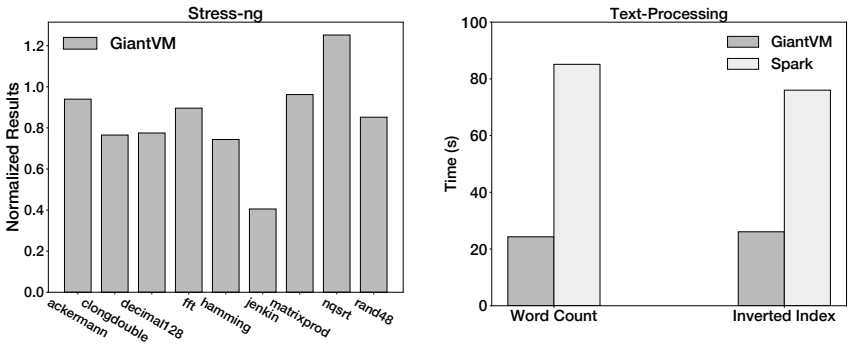
Figure 10. Scalability of CPU-intensive applications. The result of 4 vCPUs of each workload is normalized to 1.

5.3 Applications

At last, we measure the performance of various applications including CPU methods of `stress-ng` and text-processing.

Stress-ng We test with CPU methods in `stress-ng` to investigate the performance of CPU-intensive applications. The experiment of `stress-ng` is configured as (4 nodes $*$ n vCPUs), where n ranges from 1 to 8. Figure 10 shows the performance improvement when the available vCPUs increase. The results of 4 vCPUs are normalized to 1. There is linear scalability for these applications, for there is little shared data hence the page faults. The figure shows ideal applications that can be deployed on GiantVM. Figure 11 (a) shows the comparison between GiantVM and vanilla QEMU-KVM with the same 16 vCPUs. GiantVM is 1.34X slower than the baseline on average. The main overhead is timer interrupt and the submission stage of multi-thread processes. First, the timer interrupt frequency is configured as 100, namely there are 100 timer interrupts per second. Upon each interrupt, the handler of each vCPU touches the same pages, which leads to page faults. Second, the tests are conducted by iterations and results are submitted to a global view at the end of each iteration. Each iteration is usually extremely short (2.5ms-1.5μs). When the iteration ends, there will be page thrashing. But thanks to throughput-orientation of these CPU methods, the results are acceptable.

Text-processing We write two text-processing programs `Word Count` and `Inverted Index` from scratch and compare the results with equivalent Spark cluster to investigate the performance of GiantVM at the best scenario. `Word Count` does word count on a 5GB textfile and `Inverted Index` makes inverted index (a map containing `word->offset`) of



(a) Stress-ng (The higher the better)

(b) Text-processing

Figure 11. Applications results

a 100MB textfile. The code is written with moderate concern, such as allocating page-aligned malloc space. The experiment is configured as (4 nodes $*$ 6 vCPUs) and the Spark cluster is configured as 4 machines, 6 cores for each slave worker. Figure 11 (b) shows GiantVM outperforms Spark by 3.5x and 3.3x respectively. However, Spark is a fault tolerance oriented software. It sacrifices performance to support better scalability [44], especially for such a small set of data. But what we want to emphasize is that without significant performance degradation, programmers can use GiantVM to write code by their customized methods.

6 Discussion

6.1 Memory Consistency Model

Currently, GiantVM supports a strict memory model, the sequential consistency. Generally speaking, relaxing the memory model can improve the performance. On the x86 platforms, the memory model is x86-TSO [53], which can be intuitively described as:

- Writes are buffered in a local *store buffer* which is invisible to other CPUs. Reads check store buffer at first, then conduct global memory lookup if the item is missed in store buffer.
- A `mfence`[3]instruction flushes the store buffer.
- A buffered write can be broadcasted at any time.

We discuss the possibility to implement x86-TSO in DSM here. The key point is to emulate `mfence`. Generally speaking, there are three approaches to do this:

1. Binary translation.
2. Para-virtualization.
3. Future hardware support of emulatable `mfence`.

(1) is simple to emulate `mfence` but it is usually much slower than hardware-assisted virtualization. (3) depends on the future work of Intel or AMD. Thus we focus on (2). The direct idea is that adding a hypercall after every `mfence`[4]of the guest. This is impractical since `mfence` is a user-level instruction and used everywhere. For example, `mfence` is generated for operations on Java `volatile` keyword [1], as well as C++ `atomic` variables [4]. As a compromise, we only consider disposing of the kernel. This leads to a hybrid mode: pages of user level follow sequential consistency and pages of kernel level follow TSO. Also we need to track kernel pages by adding hypercalls to functions of physical page allocation (`get_free_page`) and deallocation (`free_page`). It is necessary since we use EPT rather than Shadow Page Table (SPT). Under SPT mode, we can leverage this by the Linux convention, e.g., 3G-4G memory space uses TSO for a 32-bit OS. The following multi-writer protocol has been described by previous work [12, 26]. Briefly speaking, writes can be delayed until `mfence` is executed or the timeout is fired. Each TSO node keeps a *diff* between the current version and the last consistent version. The protocol uses the vector clock to determine the order of the operations in case of write conflicts and then merges all *diffs*.

6.2 Fault Tolerance

Fault tolerance is not supported in GiantVM, i.e., one component failure leads to the failure of the whole system. However, as a hypervisor, it is inefficient to replicate the application states for GiantVM, since the hypervisor is not aware of which part of data should be replicated. Instead, the responsibility of fault tolerance should better be taken by the applications. Many applications hold the assumptions that the underlying physical machines/VMs are unreliable.

7 Related Work

Single System Image Rajkumar Buyya *et al.* conduct the comprehensive surveys of SSI before 2016 [22, 23, 34]. The

[3]Some atomic instructions flush the store buffer, too. For simplicity, we also use `mfence` to refer to these instructions.

[4]Atomic instructions can be treated as normal `mfence`, although they require a global lock in the implementation of x86 architectures. The consistency of vCPUs in the same node is guaranteed by the architecture and vCPUs in other nodes cannot read an inconsistent value.

early SSI systems include distributed OS or cluster OS [48, 52, 56]. They explore many interesting technologies like process migration, process checkpoints, shared file systems, etc. Since the 2000s, it tends to be extended based on existing operating systems such as Linux to have SSI features including Kerrighed [59] and MOSIX [15]. Recent distributed OSes include Helios [47], fos [61], Popcorn [16], and LegoOS [64]. A shortcoming of distributed OS is that it lacks driver support of heterogeneous devices such as FPGA and GPU. Most drivers of these devices are not open-source and are impractical to port on a dedicated OS.

Recently, there is an interest of connecting physical machines with high-speed network to construct SSI. FireBox [13] and The Machine [36] provide a single machine composed of numerous computing sockets and DRAMs. Memory Blade [41, 42], Infiniswap [33], and Remote Regions [10] enable local applications access memory on the remote node. ReFlex [38], vSAN [31], Decibel [45], and PolarFS [25] build up a shared storage system among multiple storage nodes.

System Virtualization Trap-and-emulate and binary translation are traditional software solutions to virtualization, and hardware-assisted virtualization has been dominant on x86 platforms [9] since the debut of Intel VT-x [58]. In addition to KVM, Xen [17] is another popular open-source hypervisor, leveraging para-virtualization techniques. Many works about system virtualization focus on I/O virtualization. QEMU provides emulated I/O devices by default. SR-IOV [30] virtualizes one physical device into multiple instances. VirtIO [51] uses a para-virtualization method, which lets the guest OS interact with virtualized devices directly.

There are also researches on heterogeneous computing and new hardware. Yazdanshenas *et al.* complete a survey on FPGA virtualization [63]. gVirt [57, 62] addresses the issue of GPU virtualization. De Lacerda Ruivo *et al.* implement Infiniband virtualization via SR-IOV [50]. Therefore, the virtualization of new heterogeneous devices such as FPGA and GPU are also feasible in our design and we will support them in a near feature.

Distributed Shared Memory Protic *et al.* [49] give a good survey of DSM researches in the 1990s. Ivy [40] is the ancestor of most modern DSM systems. Mirage [32] improves Ivy by guaranteeing page ownership for a fixed period of time to mitigate page thrashing. Munin [26] provides different pages with different coherence mechanisms. TreadMarks [12] uses *diff* algorithms to reduce network transfers. Hotpot [54], Grappa [46], and GAM [24] are the recent work of integrating DSM and RDMA.

SSI by Virtualization ScaleMP [3] proposes the *vSMP Foundation*, a *type-I* hypervisor which has similar architecture and target as GiantVM. Although there are no published academic articles, we learn some technical details from product introductions, release notes, and patents. *AnyIO* is a component which enables pass-through distributed I/O. According to our developing experiences, the positive performance of ScaleMP relies on the tuning of guest applications. A collection of tools called *vSMPPP* is asked to install on the guest OS when deploying their products. It uses a dedicated Linux kernel and user-level libraries. A typical example is that vSMPPP changes the cacheline definition of Linux kernel to 4KB. TidalScale supports similar features of ScaleMP. Its highlight is the support of the migration of vCPUs across the nodes to mitigate page thrashing [18].

vNUMA [27] is also a *type-I* distributed hypervisor on Itanium leveraged by pre-virtualization [39]. Its main contribution is the enhancement of the Ivy protocol for underlying DSM. But the optimization cannot satisfy our target. First, to implement the multiple-writer protocol, vNUMA requires the hypervisor to trap each write. It introduces ~250 cycles overhead in vNUMA, and it is much larger in a type-II hypervisor like GiantVM. Second, the optimization of atomic instructions relies on a special feature of Itanium: read operations can be reordered with atomic instructions. While x86 disallows this behavior since atomic instructions in x86 have the `mfence` semantics. Further, all type-I hypervisors (including ScaleMP and TidalScale) have the same problem that they require extensive and intrusive changes as a dedicated distributed OS, i.e., the lack of general purpose and heterogeneous devices support in this ecosystem.

8 Conclusion

In this paper, we present GiantVM, a distributed hypervisor for the SSI abstraction based on distributed shared memory, implementing simple programming model with as little additional overhead as possible. GiantVM is an open source type-II hypervisor based on the state-of-the-art QEMU-KVM and provides the virtualized CPU, memory, and I/O resources aggregated from multiple machines to guest OS. To the best of our knowledge, this is the first attempt in applying this design to Linux. Our preliminary experiments suggest that GiantVM could be a promising step towards SSI at the hypervisor and be a cornerstone for future researches.

Acknowledgements

We thank Bo Peng, Jiacheng Ma, Yanqiang Liu, and Yun Wang for their invaluable help. We also thank the anonymous and our shepherd Gilles Muller for their feedback. This work was supported in part by National Key Research & Development Program of China (No.2016YFB1000502), National NSF of China (NO. 61672344, 61525204, 61732010), and Shanghai Key Laboratory of Scalable Computing and Systems.

References

[1] 2019. The JSR-133 Cookbook for Compiler Writers. http://gee.cs.oswego.edu/dl/jmm/cookbook.html

[2] 2019. Perf. https://perf.wiki.kernel.org/index.php/Main_Page

[3] 2019. ScaleMP. https://www.scalemp.com

[4] 2019. std::atomic. https://en.cppreference.com/w/cpp/atomic/atomic

[5] 2019. Stress-ng. http://kernel.ubuntu.com/~cking/stress-ng/

[6] 2019. Sysbench. https://github.com/akopytov/sysbench/

[7] 2019. TidalScale. https://www.tidalscale.com

[8] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. 2006. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal* 10, 3 (2006), 179 – 192. http://search.ebscohost.com/login.aspx?direct=true&db=egs&AN=22445025&site=eds-live

[9] Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*. 2–13. https://doi.org/10.1145/1168857.1168860

[10] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. 775–787. https://www.usenix.org/conference/atc18/presentation/aguilera

[11] AMD. 2005. *AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual*.

[12] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Peter J. Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. 1996. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer* 29, 2 (1996), 18–28. https://doi.org/10.1109/2.485843

[13] Krste Asanović. 2014. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *FAST*.

[14] Infiniband Trade Association. 2008. *InfiniBand architecture volume 1, general specifications*.

[15] Amnon Barak and Oren La'adan. 1998. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Comp. Syst.* 13, 4-5 (1998), 361–372. https://doi.org/10.1016/S0167-739X(97)00037-X

[16] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. 2015. Popcorn: Bridging the Programmability Gap in heterogeneous-ISA Platforms. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 29, 16 pages. https://doi.org/10.1145/2741948.2741962

[17] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*. 164–177. https://doi.org/10.1145/945445.945462

[18] C Gordon Bell and Ike Nassi. 2018. Revisiting Scalable Coherent Shared Memory. *Computer* 51, 1 (2018), 40–49.

[19] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, Berkeley, CA, USA, 41–41. http://dl.acm.org/citation.cfm?id=1247360.1247401

[20] Christopher J. Berry, James D. Warnock, John Isakson, John Badar, Brian Bell, Frank Malgioglio, Guenter Mayer, Dina Hamid, Jesse Surprise, David Wolpert, Ofer Geva, Bill Huott, Leon J. Sigal, Sean M. Carey, Richard F. Rizzolo, Ricardo Nigaglioni, Mark Cichanowski, Dureseti Chidambarrao, Christian Jacobi, Anthony Saporito, Arthur O'neill, Robert Sonnelitter, Christian G. Zoellin, Michael H. Wood, and José Neves. 2018. IBM z14™: 14nm microprocessor for the next-generation mainframe. In *2018 IEEE International Solid-State Circuits Conference, ISSCC 2018, San Francisco, CA, USA, February 11-15, 2018*. 36–38. https://doi.org/10.1109/ISSCC.2018.8310171

[21] Timothy Broomhead, Laurence Cremean, Julien Ridoux, and Darryl Veitch. 2010. Virtualize Everything but Time. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 451–464. http://dl.acm.org/citation.cfm?id=1924943.1924975

[22] Rajkumar Buyya. 1997. Single System Image: Need, Approaches, and Supporting HPC Systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 1997, June 30 - July 3, 1997, Las Vegas, Nevada, USA*. 1106.

[23] Rajkumar Buyya, Toni Cortes, and Hai Jin. 2001. Single System Image. *IJHPCA* 15, 2 (2001), 124–135. https://doi.org/10.1177/109434200101500205

[24] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. 2018. Efficient Distributed Memory Management with RDMA and Caching. *PVLDB* 11, 11 (2018), 1604–1617. https://doi.org/10.14778/3236187.3236209

[25] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *PVLDB* 11, 12 (2018), 1849–1862. https://doi.org/10.14778/3229863.3229872

[26] John B. Carter, John K. Bennett, and Willy Zwaenepoel. 1991. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles, SOSP 1991, Asilomar Conference Center, Pacific Grove, California, USA, October 13-16, 1991*. 152–164. https://doi.org/10.1145/121132.121159

[27] Matthew Chapman and Gernot Heiser. 2009. vNUMA: A Virtual Shared-memory Multiprocessor. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference (USENIX'09)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=1855807.1855809

[28] David Cunningham, David Grove, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Hiroki Murata, Vijay Saraswat, Mikio Takeuchi, and Olivier Tardieu. 2014. Resilient X10: Efficient Failure-aware Programming. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*. ACM, New York, NY, USA, 67–80. https://doi.org/10.1145/2555243.2555248

[29] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[30] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. 2012. High performance network virtualization with SR-IOV. *J. Parallel and Distrib. Comput.* 72, 11 (2012), 1471–1480. https://doi.org/10.1016/j.jpdc.2012.01.020 Communication Architectures for Scalable Systems.

[31] Bryan Fink, Eric Knauft, and Gene Zhang. 2017. vSAN: Modern Distributed Storage. *Operating Systems Review* 51, 1 (2017), 33–37. https://doi.org/10.1145/3139645.3139651

[32] B. Fleisch and G. Popek. 1989. Mirage: A Coherent Distributed Shared Memory Design. *SIGOPS Oper. Syst. Rev.* 23, 5 (Nov. 1989), 211–223. https://doi.org/10.1145/74851.74871

[33] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*. 649–667. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu

[34] Philip D. Healy, Theo Lynn, Enda Barrett, and John P. Morrison. 2016. Single system image: A survey. *J. Parallel Distrib. Comput.* 90-91 (2016), 35–51. https://doi.org/10.1016/j.jpdc.2016.01.004

[35] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 185–201. http://dl.acm.org/citation.cfm?id=3026877.3026892

[36] Kimberly Keeton. 2015. The Machine: An Architecture for Memory-centric Computing. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers, ROSS 2015, Portland, OR, USA, June 16, 2015*. 1:1. https://doi.org/10.1145/2768405.2768406

[37] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07)*.

[38] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*. 345–359. https://doi.org/10.1145/3037697.3037732

[39] Joshua LeVasseur, Volkmar Uhlig, Yaowei Yang, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. 2008. Pre-virtualization: Soft layering for virtual machines. In *13th Asia-Pacific Computer Systems Architecture Conference, ACSAC 2008, Hsinchu, China, August 4-6, 2008*. 1–9. https://doi.org/10.1109/APCSAC.2008.4625458

[40] Kai Li and Paul Hudak. 1989. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst.* 7, 4 (Nov. 1989), 321–359. https://doi.org/10.1145/75104.75105

[41] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*. 267–278. https://doi.org/10.1145/1555754.1555789

[42] Kevin T. Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2012. System-level implications of disaggregated memory. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*. 189–200. https://doi.org/10.1109/HPCA.2012.6168955

[43] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network Stack Specialization for Performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 175–186. https://doi.org/10.1145/2619239.2626311

[44] Frank McSherry, Michael Isard, and Derek Gordon Murray. 2015. Scalability! But at what COST?. In *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*. https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry

[45] Mihir Nanavati, Jake Wires, and Andrew Warfield. 2017. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*. 17–33. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/nanavati

[46] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-Tolerant Software Distributed Shared Memory. In *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*. 291–305. https://www.usenix.org/conference/atc15/technical-session/presentation/nelson

[47] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. 2009. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 221–234. https://doi.org/10.1145/1629575.1629597

[48] John K. Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. 1988. The Sprite Network Operating System. *IEEE Computer* 21, 2 (1988), 23–36. https://doi.org/10.1109/2.16

[49] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. 1996. Distributed shared memory: concepts and systems. *IEEE P&DT* 4, 2 (1996), 63–71. https://doi.org/10.1109/88.494605

[50] Tiago Pais Pitta De Lacerda Ruivo, Gerard Bernabeu Altayo, Gabriele Garzoglio, Steven Timm, Hyunwoo Kim, Seo-Young Noh, and Ioan Raicu. 2014. Exploring Infiniband Hardware Virtualization in OpenNebula towards Efficient High-Performance Computing. *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2014), 943–948.

[51] Rusty Russell. 2008. Virtio: Towards a De-facto Standard for Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.* 42, 5 (July 2008), 95–103. https://doi.org/10.1145/1400097.1400108

[52] Jerome H. Saltzer, Roy Levin, and David D. Redell (Eds.). 1983. *Proceedings of the Ninth ACM Symposium on Operating System Principles, SOSP 1983, Bretton Woods, New Hampshire, USA, October 10-13, 1983*. ACM. https://doi.org/10.1145/800217

[53] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97. https://doi.org/10.1145/1785414.1785443

[54] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. ACM, New York, NY, USA, 323–337. https://doi.org/10.1145/3127479.3128610

[55] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. 2017. RFP: When RPC is Faster than Server-Bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. 1–15. https://doi.org/10.1145/3064176.3064189

[56] Andrew S. Tanenbaum, M. Frans Kaashoek, Robbert van Renesse, and Henri E. Bal. 1991. The Amoeba distributed operating system - A status report. *Computer Communications* 14, 6 (1991), 324–335. https://doi.org/10.1016/0140-3664(91)90058-9

[57] Kun Tian, Yaozu Dong, and David Cowperthwaite. 2014. A Full GPU Virtualization Solution with Mediated Pass-Through. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 121–132. https://www.usenix.org/conference/atc14/technical-sessions/presentation/tian

[58] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H. Leung, and Larry Smith. 2005. Intel Virtualization Technology. *IEEE Computer* 38, 5 (2005), 48–56. https://doi.org/10.1109/MC.2005.163

[59] Geoffroy Vallée, Renaud Lottiaux, Louis Rilling, Jean-Yves Berthou, Ivan Dutka Malhen, and Christine Morin. 2003. A Case for Single System Image Cluster Operating Systems: The Kerrighed Approach. *Parallel Processing Letters* 13, 2 (2003), 95–122. https://doi.org/10.1142/S0129626403001185

[60] Charles F. Webb. 2008. IBM z10: The Next-Generation Mainframe Microprocessor. *IEEE Micro* 28, 2 (2008), 19–29. https://doi.org/10.1109/MM.2008.26

[61] David Wentzlaff and Anant Agarwal. 2009. Factored Operating Systems (Fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Oper. Syst. Rev.* 43, 2 (April 2009), 76–85. https://doi.org/10.1145/1531793.1531805

[62] Mochi Xue, Kun Tian, Yaozu Dong, Jiacheng Ma, Jiajun Wang, Zhengwei Qi, Bingsheng He, and Haibing Guan. 2016. gScale: Scaling up GPU Virtualization with Dynamic Sharing of Graphics Memory Space.

In *USENIX Annual Technical Conference*. USENIX Association, Denver, CO, USA, 579–590. https://www.usenix.org/conference/atc16/technical-sessions/presentation/xue

[63] Sadegh Yazdanshenas and Vaughn Betz. 2017. Quantifying and mitigating the costs of FPGA virtualization. In *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*. 1–7. https://doi.org/10.23919/FPL.2017.8056807

[64] Yilun Chen Yizhou Shan, Yutong Huang and Yiying Zhang. 2018. Lego: A Decomposed, Distributed OS for Hardware Resource Disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA. https://www.usenix.org/conference/osdi18/presentation/shan

[65] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[66] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, New York, NY, USA, 523–536. https://doi.org/10.1145/2785956.2787484