GiantVM: A Novel Distributed Hypervisor for Resource Aggregation with DSM-aware Optimizations

XINGGUO JIA, JIN ZHANG, BOSHI YU, XINGYUE QIAN, ZHENGWEI QI, and HAIBING GUAN, Shanghai Jiao Tong University

We present GiantVM,[1] an open-source distributed hypervisor that provides the many-to-one virtualization to aggregate resources from multiple physical machines. We propose techniques to enable distributed CPU and I/O virtualization and distributed shared memory (DSM) to achieve memory aggregation. GiantVM is implemented based on the state-of-the-art type-II hypervisor QEMU-KVM, and it can currently host conventional OSes such as Linux. (1) We identify the performance bottleneck of GiantVM to be DSM, through a top-down performance analysis. Although GiantVM offers great opportunities for CPU-intensive applications to enjoy the aggregated CPU resources, memory-intensive applications could suffer from cross-node page sharing, which requires frequent DSM involvement and leads to performance collapse. We design the guest-level thread scheduler, DaS (DSM-aware Scheduler), to overcome the bottleneck. When benchmarking with NAS Parallel Benchmarks, the DaS could achieve a performance boost of up to 3.5×, compared to the default Linux kernel scheduler. (2) While evaluating DaS, we observe the advantage of GiantVM as a resource reallocation facility. Thanks to the SSI abstraction of GiantVM, migration could be done by guest-level scheduling. DSM allows standby pages in the migration destination, which need not be transferred through the network. The saved network bandwidth is 68% on average, compared to VM live migration. Resource reallocation with GiantVM increases the overall CPU utilization by 14.3% in a co-location experiment.

CCS Concepts: • **Computer systems organization → Cloud computing;**

Additional Key Words and Phrases: Resource aggregation, virtualization, single system image, distributed shared memory, false sharing

ACM Reference format:
Xingguo Jia, Jin Zhang, Boshi Yu, Xingyue Qian, Zhengwei Qi, and Haibing Guan. 2022. GiantVM: A Novel Distributed Hypervisor for Resource Aggregation with DSM-aware Optimizations. *ACM Trans. Archit. Code Optim.* 19, 2, Article 20 (March 2022), 27 pages.
https://doi.org/10.1145/3505251

20

[1]This article is an extension of a conference article at VEE'20 [67]. We add DSM-aware optimizations to GiantVM, including DaS in the guest OS, and running Barrelfish as a substitute for Linux guest OS. We have a top-down evaluation for GiantVM. We observe the advantage of GiantVM as a resource reallocation facility, enabling lower network bandwidth consumption compared to VM live migration.

This work was supported in part by National NSF of China (NO. 61732010) and Shanghai Key Laboratory of Scalable Computing and Systems.
Authors' address: X. Jia, J. Zhang, B. Yu, X. Qian, Z. Qi, and H, Guan, Shanghai Jiao Tong University, 800 Dongchuan Rd, Minhang District, Shanghai, 200240, China; emails: {jiaxg1998, jzhang3002, 201608ybs, qianxingyueyx6, qizhwei, hbguan}@sjtu.edu.cn.

1544-3566/2022/03-ART20 $15.00
https://doi.org/10.1145/3505251

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 [26] and Spark [66] quickly become prevalent. Today, the conventional wisdom is that traditional scale-up solutions such as mainframes [63] 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. To overcome this dilemma, the **Single System Image** (**SSI**), which runs a single **Operating System** (**OS**) instance on a cluster [20], has been proposed to hide complex distributed programming models. However, the traditional SSI usually is a distributed OS or cluster OS [53]. 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. Cloud providers could provide a virtual machine to achieve simpler management and better resource utilization, without significant modifications to existing software.

To exploit both SSI and virtualization, previous attempts have been made to combine them based on **Distributed Shared Memory** (**DSM**) [44, 54]. 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 [40], while TidalScale [44] needs a dedicated HyperKernel. Similar to the distributed OS, these customized type-I hypervisors are difficult to deploy and maintain on the cloud.

GiantVM, first published in [67], builds an easy-to-use SSI platform based on the state-of-the-art type-II hypervisor QEMU-KVM for aggregating computing, memory, storage, and I/O resources on which the general guest OS can run. Figure 1 shows its architecture, which can host Linux and other experimental OSes like sv6 [23], xv6 [24], and Barrelfish [13]. The distributed CPU virtualization works independently on each node at most times, except for the inter-core communication between two nodes, on which we should focus. For the distributed I/O virtualization, a primary node is designated, by which I/O requests are multiplexed. The memory virtualization is a DSM based on Ivy [41].

In the evaluation part, we analyze the performance of GiantVM in a top-down approach. First, we run Sysbench [39] and Stress-ng [36] microbenchmarks to make the performance bottleneck assumptions from the guest side of GiantVM, i.e., the interrupt and I/O forwarding overhead are negligible, while DSM overhead is significant. To test our assumptions, we profile the time spent on each part of GiantVM when running representative workloads, identifying the performance bottleneck to be DSM. We also have a comparison between TCP and RDMA network backends of GiantVM, claiming that the RDMA network is crucial for GiantVM to run efficiently.

The ease of programming comes at a cost, i.e., the DSM component overhead. The software above GiantVM hypervisor is unaware of the underlying distributed system, causing performance issues. Two DSM-aware optimizations are proposed to turn GiantVM into a practical system for use. We substitute the Linux guest OS with Barrelfish [13] to minimize kernel-level false sharing and implement the **DSM-aware Scheduler** (**DaS**) for the Linux guest to minimize cross-node page sharing, through run-time thread scheduling. Both techniques aim at reducing the DSM involvement. A kernel-intensive web server achieves 6.4× throughput when running on Barrelfish, compared to Linux. **NAS Parallel Benchmarks** (**NPB**), (OpenMP version) [9] running in DaS could gain a performance boost of up to 3.5×, compared to running in the Linux ***Completely Fair Scheduler*** (**CFS**) [2].

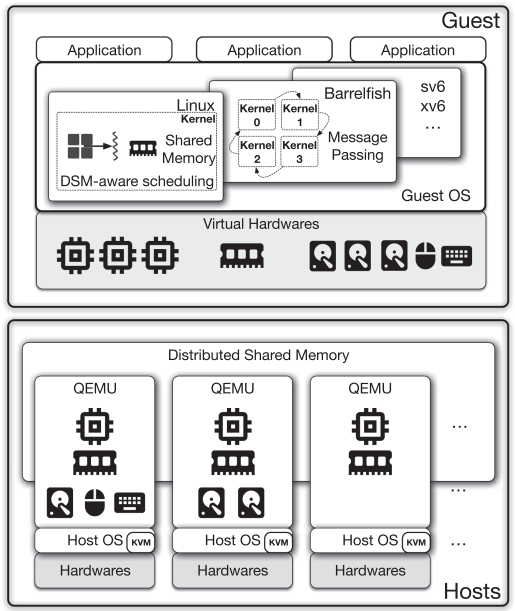

Fig. 1. The architecture of GiantVM.

While evaluating DaS, we observe that GiantVM could migrate tasks across nodes efficiently and economically, simply by guest-level thread scheduling. Migration with GiantVM addresses the residual dependencies problem, which is difficult to deal with for process migration [45]. Meanwhile, since the DSM component of GiantVM allows standby pages in the migration destination, only a small portion of guest pages (about 1/3 in our experiment) is supposed to transfer in the network vis-á-vis VM live migration [22], reducing 68% of network bandwidth consumption. We design a **Load-aware Scheduler** (**LaS**) in the guest OS, to balance the CPU load among the cluster. As a novel approach to address the CPU underutilization in data centers, GiantVM helps increase the overall CPU utilization by 14.3% relatively in a co-location experiment.

The key contributions of this article about GiantVM and its optimizations can be summarized as follows:

— We propose GiantVM, the first distributed type-II hypervisor, which provides 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, which overcomes the difficulties of sharing CPU, memory, and I/O devices across machines.
— We evaluate GiantVM in a top-down approach, claiming that RDMA could be the key enabling technology for GiantVM, and identifying GiantVM's performance bottleneck to be DSM.
— We propose two DSM-aware optimizations to turn GiantVM into a practical system to use, overcoming the DSM bottleneck. Also, GiantVM achieves additional benefits for migrating tasks, compared to VM live migration.

The rest of this article is organized as follows. Section 2 describes the background of GiantVM. We elaborate on the design and implementation of GiantVM, DaS, and LaS in Sections 3 and 4,

respectively. Section 5 gives a top-down evaluation of GiantVM and the results of DSM-aware optimizations. Sections 6 and 7 present the discussion and related work, respectively. Finally, we conclude our work and present future work in Section 8.

2 BACKGROUND

System Virtualization and QEMU-KVM: System virtualization refers to an abstraction layer that separates the physical hardware from OSes to gain more flexible resource utilization, enabling 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 [69] and AMD [70] both have their hardware virtualization solutions [16]. We mainly focus on Intel's technologies in this article.[2]

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, thus suffering less overhead. While a type-II hypervisor relies on a host OS, reusing the tools provided by the host OS, for example, QEMU-KVM. KVM [37] is a Linux kernel module that supports hardware-assisted virtualization, reusing Linux kernel functionalities. QEMU [15] 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, while it handles the emulation of I/O devices.

Distributed Shared Memory and False Sharing: DSM [51] 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.

In DSM, the phenomenon that two processors access the same page is called page sharing, including true and false sharing. True sharing is two threads accessing the same variable, while false sharing is because they each access a different variable on the same page. There is a tradeoff between large *coherence blocks*[3] to enhance spatial locality and small ones to reduce false sharing. DSM systems have large coherence blocks (4KiB), adding to the severity of false sharing. Efforts are made to reduce false sharing, such as Sheriff [43], isolating shared data updates to separate physical addresses, and relaxed memory consistency models [19]. These solutions may either work for particular applications, or require great engineering efforts.

RDMA: RDMA is a recent network technology. RDMA capable networks such as **InfiniBand** (**IB**) usually provide low latency and high throughput compared to the traditional Ethernet and TCP/IP combination [8]. The average latency of TCP is 9× higher than RDMA [68]. 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 the network socket. One-sided RDMA usually has lower latency

[2]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 EPT entry), a special VMExit called *EPT violation* will be triggered.

[3]In a shared-memory system, shared data is replicated into the local cache/memory of the processor accessing it to increase data reference locality. To ensure coherence between local and shared data, memory is grouped into *coherence blocks*. For the hardware cache, a coherence block is a typically 64-byte cache line. For DSM, a coherence block is a 4KiB page. A write to one local block leads to the invalidation of all other copies of this block. A read to an invalid local block leads to a cache miss/page fault bringing the lastest copy of the block (written by the lastest writer) to the local cache/memory.

than two-sided RDMA. However, the opposite conclusion can be drawn when one transaction can be completed by either multiple one-sided operations or a single two-sided operation [57].

Resource Reallocation in Data Centers: Underutilization is well discussed in data centers [21]. After the job scheduler dispatches a task to a node, its CPU usage fluctuates over time. Alibaba Cluster Trace reports 26.5% and 38.2% average CPU utilization in 2017 and 2018 [5], respectively. Tasks could be classified into **latency-critical** (**LC**) and **best-effort** (**BE**) types [21]. LC tasks have a strict demand for **Quality of Service** (**QoS**). Thus the job scheduler allocates enough resources to them. As a result, a high surplus utilization is left unexploited. Due to the heterogeneity and variability of CPU utilization, the job scheduler performs resource reallocation at run-time, i.e., tasks are migrated among the cluster. BE tasks co-locate with LC tasks when CPU usage is low, and are migrated away when the CPU usage of LC tasks soars.

Techniques enabling resource reallocation include process migration [45] and container/VM live migration [22, 46], to migrate the low-priority BE tasks. Process migration has difficulties dealing with residual dependencies, i.e. the data structures that break through the process boundaries, thus is rarely used. Container live migration suffers a much longer downtime than VM live migration [47]. Transferring all guest memory pages through the network is the performance bottleneck for VM live migration, which consumes precious bandwidth, as the frequency of migration significantly rises. Consequently, network-intensive tasks may suffer from insufficient bandwidth.

3 DESIGN

3.1 Principles and Overview

- **Principle-1:** *Transparency from applications.* We aim at implementing 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.
- **Principle-2:** *A type-II hypervisor built on the x86 platforms.* First, although the 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 in these areas is designed for x86 platforms. Of course, it is feasible to apply our approach to other architectures.
- **Principle-3:** *A dedicated hypervisor rather than kernel.* The type-II hypervisor is usually lightweight 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.

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 in providing a single consistent view of virtualized hardware for the guest, we distinguish local and remote resources. Each hypervisor instance on different physical machines creates a complete virtual machine image using the same parameters, but 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.

3.2 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, except 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, 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 the local vCPU, we just process it as usual; otherwise, we send it to the remote hypervisor instance and inject it into the APIC of the remote vCPU.

3.3 Distributed Memory

Memory virtualization is the most critical part of GiantVM. Since the memory model of x86 platforms is x86-TSO [55], 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. Based on the description above, we achieved it through a *sequential consistency DSM* based on the protocol of Ivy [41].

3.3.1 DSM in GiantVM. There are three states in the DSM, namely `Modified`, `Shared`, and `Invalid`. Each 4 KiB page can be in one of the three states. If it is in `Modified` state on node N, N has the exclusive ownership of this page and can read and write at will. If it is in `Shared` state, N shares the page with other nodes and the page is write-protected. If it is in `Invalid` state, the page is marked not present in the page table and N must ask other nodes for the latest copy 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 a manager responsible for a portion of guest memory space, which matches nicely with NUMA. Inspired by the similarity between NUMA and DSM [34], we notify the topology of the physical cluster to the guest OS via the NUMA topology.

When a read occurs on an `Invalid` page, or a write occurs on a `Shared` or `Invalid` page, a page fault (which we name *DSM page fault*, ***DSM PF*** for short) is triggered, and the execution is transferred to the hypervisor instance (*VMExit*). Then a request to the manager of this page is issued (*Message-1*), the manager then forwards the request to the owner of this page (*Message-1-non-owner*), or it processes the request immediately if it is already the owner. For *read* requests, the requesting node is added to the copyset of the owner and is sent to the latest copy of this page (*Message-2-read*), through the manager (*Message-2-read-non-owner*). For *write* requests, the old owner invalidates all the holders of this page, and then transfers the ownership and the latest copy to the requesting node (*Message-2-write*), through the manager (*Message-2-write-non-owner*). 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 (*VMResume*).

As discussed above, a DSM PF can lead to 2 (*Message-1, Message-2-*$*$) and 4 (*Message-1, Message-1-non-owner, Message-2-*$*$*, Message-2-*$*$*-non-owner*) messages for an *owner* and *non-owner* access, respectively (excluding the messages to invalidate obsolete copies for write requests,

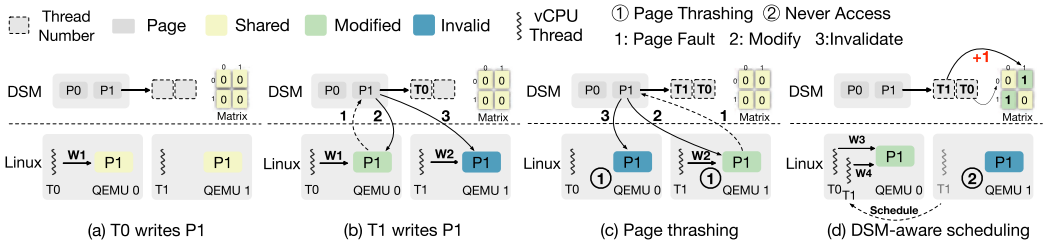

Fig. 2. DSM-aware scheduling process in the Linux guest OS. Thread 0 and 1 cause page faults on the same page one after another (in (a) and (b)), leading to page thrashing (in (c)). Then positions `(0, 1)` and `(1, 0)` in the DSM matrix are incremented by 1. Then the DaS moves thread 1 to QEMU 0 (running on Node 0). Page thrashing is therefore eliminated (in (d)).

$* = \{read, write\}$). Additionally, a VMExit/VMResume pair to transfer execution to and from the hypervisor is required, which can be costly. A vast majority of modern applications use *peer-threads* that frequently access a shared address space, i.e., they are memory intensive. DSM page faults can thus be extremely frequent for them, leading to performance collapse.

3.3.2 Detecting Page Sharing with DSM Matrix. It takes great engineering effort to mitigate false sharing by dynamic detection, compiler enhancement, and application tuning [43], or relaxing memory consistency models [19]. However, it is shown that thread scheduling, combined with additional system status information, can mitigate contention for shared resources.[4] [25, 61] Also, it requires no application modifications and is easy to integrate into current systems.

NUMA balancing is a technique [61] based on thread grouping and page migration to minimize remote NUMA-node memory accesses. It captures memory accesses with *NUMA-hinting* page faults, then migrates pages to the NUMA node that most frequently accesses them, and schedules memory-sharing threads to the same NUMA node. However, this technique does not take the underlying DSM into account. Page migration is heavy-weight on a single machine [62], and can be even more expensive on GiantVM. Also, thread grouping could happen only after a relatively long time accumulation of memory access information. Consequently, short tasks on GiantVM could suffer from DSM.

We propose the DaS based on the *DSM matrix*, avoiding modification to the NUMA balancing mechanisms and the CFS [2] in the Linux kernel. We allocate a private DSM matrix for each *process* when it starts to run in the Linux guest OS, which stores the number of *guest page faults*[5] that each thread-pair causes on the same guest page, representing the number of shared-page accesses between the thread-pairs. As shown in Figure 2(a) and (b) (W stands for *write*), each time a pair of threads cause two-page faults on page P1, the faulting page number and two thread numbers are collected to update the DSM matrix. For each memory page, we record the two threads that *most recently* caused page faults on it (the dotted-line squares in Figure 2). Then, the position corresponding to this thread-pair in the DSM matrix is incremented by 1 (+1 in Figure 2(d)), indicating that two threads *communicated* once.

[4]In our case, processes running on GiantVM contend for the shared memory provided by DSM, which contend for network bandwidth, in essence.

[5]By guest page faults, we mean the page faults triggered by guest applications, while the DSM page faults in the above section are due to the DSM protocol design. Essentially, guest page faults indicate that the guest application accesses a guest virtual page. DSM page faults are triggered by two nodes of GiantVM accessing the same guest physical page, and at least one access is a *write*. The two categories of page faults should not be confused.

3.3.3 DSM-aware Thread Scheduling Algorithm. We take the DSM matrix obtained in Section 3.3.2 as input and generate the DSM-aware scheduling decision in Algorithm 1. Our purpose is to minimize page sharing across nodes, thus reducing the network overhead of GiantVM. The approach can be explained as generating *thread cliques* within a single process and scheduling each clique to a NUMA node. At the beginning of the algorithm, each thread of P forms a clique, and we initialize C on Line 1. We merge cliques until the number of cliques is exactly N. When selecting a clique to merge with, we find the most frequently page-sharing clique using data from the DSM matrix, on Line 7. Finally, we assign cliques to NUMA nodes on Line 12, depending on the total vCPU load of each NUMA node.

ALGORITHM 1: DSM-aware scheduling algorithm

Input: P: Process. $matrix$: DSM matrix for process P. N: Number of cliques to generate.
Output: A scheduling decision for threads $t_1, \ldots, t_n$ of process P.

```
1  C ← {{t_1}, {t_2}, ..., {t_n}}, t_i ∈ P
2  // Merge threads into N cliques and store in C.
3  while |C| > N do
4      C_temp ← φ
5      while C ≠ φ do
6          clique ← C.pop()
7          clique* ← find_neighbor(C, clique, matrix)
8          C_temp.push(clique* ∪ clique)
9          C.remove(clique*)
10     C ← C_temp
11 // See Section 4.2.2 for details of CPU load balancing
12 assign_cpus(C)
```

DaS allows threads within a clique to run in the same NUMA node. We have an example of clique generating in Section 4.2.2, showing how Algorithm 1 efficiently does clique generation. When setting N to 1, CPU could become the performance bottleneck for P, but all memory accesses would be local. For $N > 1$ settings, DaS generates N cliques for P, thus P could enjoy more CPU resources, but suffer from page thrashing. Section 5.2 elaborates more on this tradeoff.

3.3.4 Migration of Tasks with GiantVM. Although GiantVM is motivated by resource aggregation, i.e., making resources of multiple nodes easily available by a single-machine application, it naturally brings additional benefits for migrating tasks. Thanks to Principle-1, applications running on GiantVM could treat the cluster as a single machine. Consequently, migration could be simply done by guest OS level scheduling, needing no additional engineering effort. Together with the ease of migration, the cost of migration could be low. While functionally equivalent to VM live migration (in terms of mitigating the QoS violation), job scheduling with GiantVM **avoids** transferring the following pages:

— *Guest unmapped physical pages.* These pages are not transferred since vCPUs never touch them, and no page state changes happen. While VM live migration has to transfer them, since the hypervisor has no guest virtual page mapping information, except the hypervisors equipped with para-virtualization support.
— *Application unused pages.* These pages are mapped into guest applications' virtual address space but never touched, due to the ubiquitous data locality. Even para-virtualization technologies could not discern them.

— *Compiler/application-defined read-only data.* GiantVM allows multiple copies of `Shared` pages to co-exist on multiple nodes. They could hold compiler-defined read-only data such as `.rodata` and `.text` segments, and application-defined read-only data like the text for **`WordCount (WC)`**. They are not transferred across nodes, except for the initial cold misses.

Also, even pages frequently written by tasks are transferred on demand, when a remote DSM PF happens. Since migrating back and forth is common in resource reallocation, cold misses for read-only data are also eliminated. We claim that GiantVM is a better design tailored for resource reallocation, compared to VM live migration.

We propose the LaS based on GiantVM. BE tasks run on GiantVM, scheduled across NUMA nodes by LaS. Principle-2 allows[6] LC tasks to run on physical hosts, in a cluster equipped with GiantVM. When they have a low CPU requirement, LaS co-locates BE tasks with them on the same node. When the CPU usage of LC tasks bursts, LC tasks suffer a potential QoS violation, thus LaS schedules BE tasks away from them, to a lower-loaded node. QoS of LC tasks is guaranteed while surplus CPU resource is used for BE tasks.

3.4 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 the initiative to have all I/O requests and interrupts of a disk device processed normally. Furthermore, 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. However, it is possible to assign different devices to different nodes for the sake of load-balancing.

Our idea is straightforward. First, whenever a device generates an interrupt, 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 [67] is based on QEMU 2.8.1.1 and KVM (Linux kernel v4.9.76) on the x86 platform with around 7 K lines of code. The hypervisor instance is the modified QEMU-KVM hypervisor running on each node. DaS is implemented as a Linux kernel module for the guest OS with around 1.4 K lines of code. LaS is a Bash script running on the guest OS user space, with around 200 lines of code.

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, described in Sections 4.1 and 4.3. For memory, we implement a DSM module in KVM and adapt QEMU for it, described in Section 4.2. DaS requires only thread scheduling APIs provided by the Linux kernel, thus needing little guest OS modification. LaS needs no modification to both the guest kernel and the applications. All the mechanisms we implement to support GiantVM is shown in

[6]Type-I distributed hypervisors like TidalScale [44] and ScaleMP [54] disallow tasks to run on hosts since they are directly installed on the bare-metal.

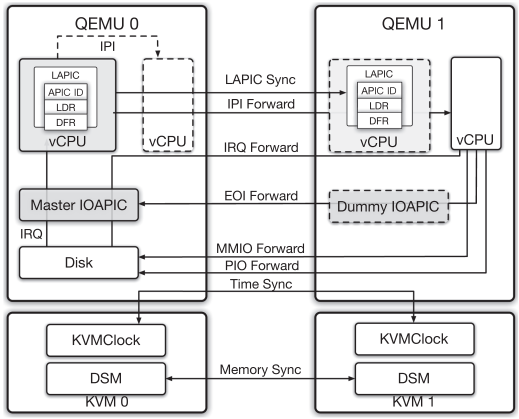

Fig. 3. The inside of GiantVM.

Figure 3. 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 (Section 2), only two-sided RDMA is used at this time.

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 (actually all interrupts) are implemented as broadcast messages under the hood. Upon receiving 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 the 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, since it can take advantage of the memory mapping information maintained by KVM and modify the EPT easily when a DSM page status is changed. Also, the interaction with the memory management module of the 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.

4.2.1 Mapping Management. In the presence of hardware virtualization, the mapping of memory is not as simple as in traditional DSMs. First, QEMU allocates the memory for the guest in its address space, i.e., **Host Virtual Address** (**HVA**) space. Then it is registered to KVM, and KVM maintains the GPA-to-HVA mapping. When the guest accesses a memory location, it first uses the guest page table to translate GVA to GPA and then uses EPT to translate GPA to HPA. There

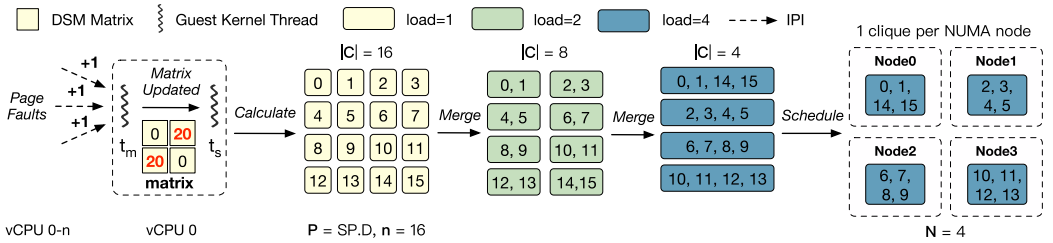

Fig. 4. Cliques generation with DaS in the Linux guest, for SP.D. DaS first waits for the DSM matrix to be significantly updated. Then, it merges threads into $N = 4$ cliques, and schedules cliques to NUMA nodes.

are some important facts: (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 directly manipulate the host page table. (2) The memory is allocated in the HVA space. 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. Furthermore, since we transfer GPA in DSM messages, we need to translate the GPA back to HVA to 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. Therefore, additional data structures are necessary for tracking *host* page status and keeping the reverse mapping from host page to guest pages.

4.2.2 DSM Matrix and DaS. Two design choices are considered when collecting data for the DSM matrix to detect page sharing. The first is to collect the faulting *GPA* in the host KVM module, by annotating the DSM PF handler. However, this GPA is only associated with vCPU causing the DSM PF, which is meaningless for DaS to do guest-level thread scheduling. The second choice is to collect guest page faults and allocate DSM matrices in the Linux guest, where we could obtain the thread information associated with the faulting *GVA*. This could be easily done by annotating the guest page fault handler `handle_pte_fault`; thus, we choose to implement from the guest side. A guest kernel thread t_m (matrix thread) running on vCPU 0 handles +1 requests for the DSM matrix (via IPI) from application threads, to avoid false sharing of the DSM matrices between NUMA nodes.

Figure 4 illustrates the DSM-aware scheduling process for application `SP.D` with 16 threads in a 4×4 GiantVM (running on 4 nodes, each hosting 4 vCPUs). A scheduler thread t_s runs Algorithm 1 on vCPU 0 together with t_m, reading the DSM matrices and calculating the scheduling decision for each process. One clique is generated for each NUMA node before Line 12. After scheduling the cliques to NUMA nodes on Line 12, each NUMA node has nearly the same number of runnable threads, ensuring inter-node CPU load balance. When implementing Line 12, we assign threads to the NUMA node of its clique using `sched_setaffinity` calls, reusing the CFS for intra-node CPU load balancing.

To avoid thread thrashing, t_s checks whether the DSM matrix is significantly updated[7] every 6,000 ms, and if true, runs Algorithm 1. To improve the accuracy of page sharing detection, we add extra *DSM-hinting* page faults in the same way as NUMA balancing, by unmapping process memory.[8] At the beginning of the 6,000 ms period, t_s marks a small portion of address space mapping invalid to get *DSM-hinting* page faults and resets all items of the DSM matrix to 0. On Line 12, only if the scheduling decision has major differences with the previous one, t_s will run Algorithm 1 at the end of the 6,000 ms. As a result, for applications with distinct execution phases, DaS could promptly detect memory access pattern changes while avoiding thread thrashing.

4.2.3 LaS for Resource Reallocation. Unlike DaS, which requires kernel-level page fault information, LaS only schedules the whole process among NUMA nodes, and we could reuse user-level interfaces to accomplish this work. We run the `taskset` command to schedule a whole process to a single NUMA node, and write to the `proc` file system to set affinities of interrupts and some workqueues. Only vCPUs on one NUMA node of GiantVM run simultaneously, thus eliminating DSM performance penalties. For LaS to get access to the host CPU load from the guest, the `proc` file system provides `steal time`, which indicates the difference between hypervisor promised CPU time and real given CPU time. A high `steal time` means a high CPU load on the host. Periodically (every 10s), LaS checks the `steal time` of the currently scheduled NUMA node. If it exceeds a pre-defined threshold (100), all OS activities, including tasks, interrupts, and workqueues, are scheduled to the NUMA node with the lowest `steal time`. We do not care about process thrashing since we run BE tasks in GiantVM.

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 3.2. The interrupt forwarding process for IOAPIC has been shown in Figure 3. 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 correctly. Then, all we need is to forward the interrupt to remote vCPUs when an MSI is generated by this device.

[7]We check by $u = \sum_{d \in M} |d - d_{old}| \geq |M|$, where d_{old} is the item of DSM matrix M obtained at the end of the previous 6,000 ms. $|M|$ is the number of items in M. This might not be accurate but could detect memory access behavior variations promptly.

[8]This does not introduce much memory footprint and overhead, since it only introduces two writes (one for marking PTE invalid and one for unmarking) to the guest page table, which typically occupies only several pages.

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.

5 EVALUATION

The evaluation of GiantVM tries to answer the following questions:

- **Question-1:** What are the strengths/weaknesses of GiantVM, in terms of CPU, memory, and I/O virtualization? What is the performance bottleneck, software or hardware? (Section 5.1)
- **Question-2:** What are the results of DSM-aware optimizations? Can these optimizations offset the price of programming ease? (Section 5.2)
- **Question-3:** What are the benefits of GiantVM over VM live migration, in terms of resource reallocation? How much does the CPU utilization increase in the cluster with the migration using GiantVM? (Section 5.3)

Setup: We run *GiantVM* on a 4-node cluster. Each node has an 8-core Intel Xeon E5-2620 v4 CPU, 128 GiB RAM, 56 Gbps Mellanox ConnectX-3 InfiniBand, and Broadcom NetXtreme BCM5720 Gigabit Ethernet. The *non-distributed QEMU-KVM* runs on a stand-alone node as the baseline, with the same number of vCPUs as GiantVM. The stand-alone node has a 24-core Intel Xeon E5-2620 v4 CPU and 128 GiB RAM. Hyper-threading is disabled on all machines. Hosts and all guests run Ubuntu 16.04 OS (kernel v4.9.76). All guests are allocated 64 GiB of memory.

In our discussion, we use $m \times n$ to denote the configuration of GiantVM with m nodes, n vCPUs on each node. All applications are allocated with sufficient memory. Hence, there is no swapping. All results are the average of five runs. The `monospaced` font is used when describing the software for experiments. All workloads in the guests run the same number of threads as guest vCPUs, unless otherwise specified.

5.1 A Top-down Performance Analysis

In this section, we study the virtualization overhead of GiantVM compared to non-distributed QEMU-KVM, and answer Question-1. We run microbenchmarks in the guest to analyze performance in the top-level abstraction. Then we do a run time breakdown analysis to reveal the performance bottleneck. Finally, we have a comparison between RDMA and TCP network backends.

Distributed vCPU: We run CPU stress methods of `Stress-ng` [36] v0.05.23 to evaluate distributed vCPU performance. The total throughput of all threads are measured (Ops/s). These methods either have a small memory footprint or only involve local memory accesses, thus the DSM involvement is minimized. Figure 5 (left) indicates that all CPU stress methods present linear scalability. Performance keeps rising even more than 8 vCPUs are required, which is the number of CPUs on each node of the cluster. GiantVM aggregates CPU resources well for workloads with infrequent memory accesses. Figure 5 (right) reports the cost of CPU resource aggregation, comparing 4×4 GiantVM and 16-vCPU QEMU-KVM. GiantVM is 1.34× slower than the baseline on average. The slowdown can be attributed to the DSM component. These tests are conducted by iterations, and results are submitted to a global view at the end of each iteration, incurring DSM overhead as all nodes require the global results. Nevertheless, the results are acceptable. These are ideal workloads that could benefit from GiantVM CPU resource aggregation.

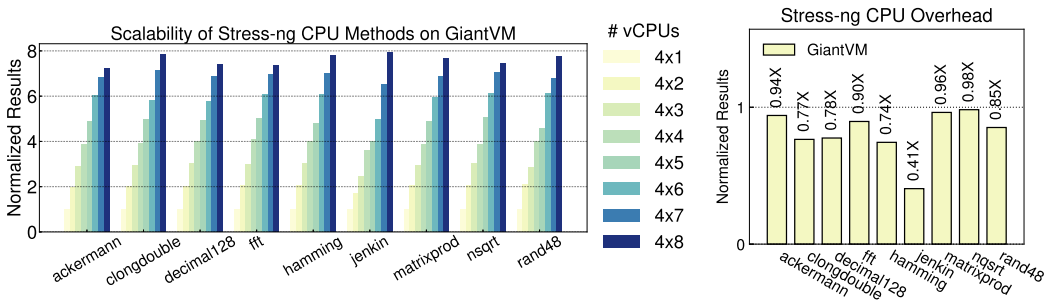

Fig. 5. Stress-ng CPU methods scalability (on the left) and overhead (on the right) on GiantVM (the higher the better). In the left figure, 4-vCPU GiantVM results are normalized to 1. In the right figure, 16-vCPU vanilla QEMU-KVM results are normalized to 1.

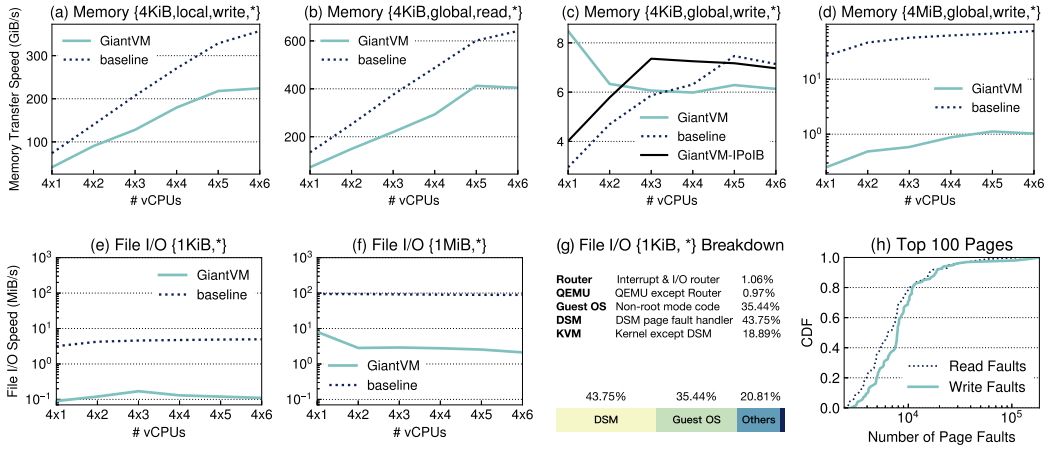

Fig. 6. Sysbench memory and file I/O results. DSM becomes the performance bottleneck of both memory and I/O intensive workloads.

Distributed Memory and I/O: We run memory and file I/O tests of Sysbench [39] v1.0.19 to evaluate distributed memory and I/O virtualization performance, respectively. Since our standalone machine has 24 CPUs, we could only run up to 4×6 configurations for GiantVM. The memory tests are configured as {*size*, *scope*, *oper*, *mode*}, doing *oper* (read or write) in *scope* (local or global) memory on *mode* (GiantVM or baseline). Local means each thread accesses its own page-aligned memory space, while global means all threads access global shared memory. The I/O tests configurations are {*size*, *mode*}, *sequentially* reading/writing *one size* (1 KiB or 1 MiB) file on *mode* (GiantVM or baseline).

The total memory/file read/write speed of all threads is reported in Figure 6(a–f). In almost all tests, baseline outperforms GiantVM significantly, especially for file I/O tests (at most 45.2× slower than single-node baseline). The results indicate a high overhead to provide a continuous address space crossing multiple nodes. For accessing local data or reading global data, the MSI protocol of DSM allows all the processors to make progress at full speed. While for writing global data, especially the file I/O workloads, which modifies data structures of the deep storage stack in the monolithic kernel, the DSM involvement is frequent. A DSM PF has a penalty of ~15*us*, while the

cache system miss is only ~60ns. The workloads frequently doing writes lead to excessive page faults, thus a great slowdown is determined.

To further investigate the slowdown of file I/O tests, we use `perf` to profile the time spent on each component of the system, and draw CDF of page fault numbers on the top-100 frequently page-faulting pages, in Figure 6(g) and (h). Only 35.4% of the time is for doing useful work, i.e., running the Guest OS. The DSM component occupies 43.8% of the time, which is catastrophic. The kernel storage system fully considers the single-machine cache system but not the GiantVM DSM system. Several locks and shared data are allocated on the *same* page, resulting in false sharing. The CDF of the number of page faults also reveals the issue. Total 70% of DSM page faults happen on pages totaling 0.5 MiB. While for interrupt and I/O forwarding, i.e., Router part in Figure 6(g), has a negligible run time of 1.06%. Thus, distributed I/O could not be the performance bottleneck.

An anomaly is the $\{4KiB, global, write, *\}$ memory test, in which we observe the "*slower is faster*" phenomenon. There exists some data that IPoIB > RDMA > baseline (DRAM & SRAM). Due to the throughput orientation[9] of the memory test, one thread could read/write a local page at full speed, before an `Invalid` message is sent from another node. Slow messages could leave more time for useful work, while fast ones could lead to *livelock*. Slow messages match the design philosophy of a weaker memory model [19], i.e., synchronize data across nodes lazily. We discuss the possibility of relaxing the memory model to be x86-TSO [55] in Section 6.1.

The phenomenon disappears when we test $\{4MiB, global, write, *\}$. The DSM now suffers from a large working set, while it is sparse enough for the cache system, and there is little cache line thrashing and livelock. At this time, a huge gap in accessing speed between L3 cache (or DRAM) and remote memory is shown.

Run Time Breakdown: We implement three big-data workloads for Linux in C++. Then we profile their run time using `perf` on a 2-node GiantVM, in the same way for `Sysbench` file I/O. These workloads represent typical big-data workloads that could benefit on GiantVM, for resource aggregation and ease of programming. Thus, we could learn the performance bottleneck of GiantVM for big-data applications. Figure 7(a–c) gives the results.

- `WC` reads a 5 GiB text file in the experiment, representing large-scale text processing workloads. It runs one thread on each CPU, and text files are divided equally among threads.
- `Pi` is a *Monte Carlo*-based π calculator, which does 20 billion random samplings, showing similar patterns to CPU-heavy workloads such as scientific computing. `Pi` creates one thread on each CPU.
- **PageRank** (**PR**) is representative of iterative tasks seen in data analysis and machine learning fields. PR works on a graph with 576 K nodes and 5.1 M edges and executes 20 iterations on it. It spawns one thread on each CPU, and we divide nodes equally for each thread.

(1) In all configurations, the *Router* overhead is negligible (<1% on average), since interrupt and I/O forwarding involve only several bytes of network transfer, containing register states. **(2)** `WC` and `PR` do a great amount of disk read for text file and graph data. Consequently, *QEMU* device emulation takes up an observable part of run time, 21.6% and 25.0% for `WC` and `PR`, respectively. **(3)** *KVM* overhead could only be seen in 2×8 PR, which is 2.02% of run time. **(4)** For `WC` and `Pi`, since they are either CPU intensive or only involve mostly local memory accesses, their DSM

[9] *Throughput orientation* means the writer can make progress without waiting for the `Invalid` message. The opposite is the *latency orientation*. For example, we must wait for the `Invalid` message from the lock-releasing thread before grabbing a lock. Throughput-oriented workloads achieve good performance if they have many CPUs to run on, while latency-oriented ones rely on low-latency `Invalid` messages to run fast.

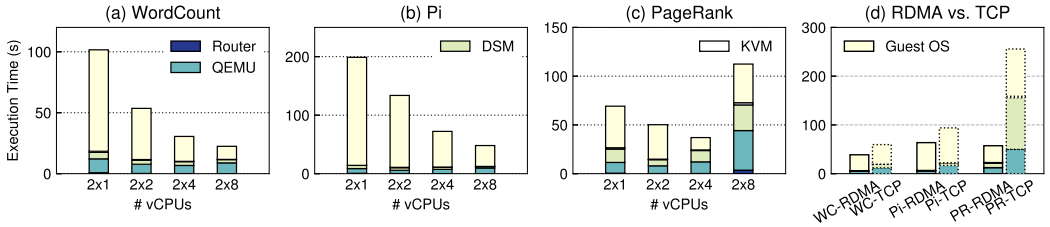

Fig. 7. Run time breakdown of typical big-data workloads on GiantVM. DSM is the performance bottleneck of GiantVM.

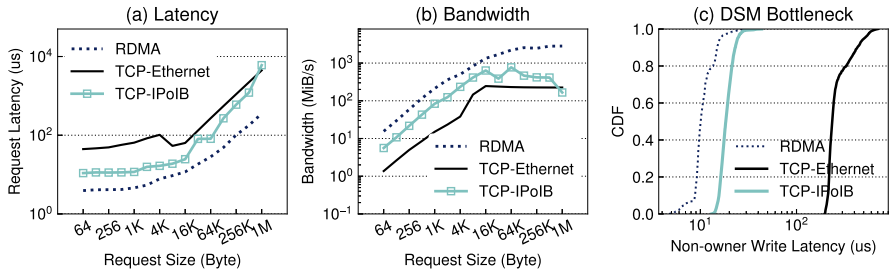

Fig. 8. The low-level comparisons among three GiantVM network backends, including latency and bandwidth comparisons.

overheads are relatively small, which are 8.4% and 4.0% on average. PR has poor memory access locality, thus incurs significant DSM overhead.

For 2 × 8 PR, the scalability issue occurs. DSM takes up to 31.3% of the execution time. This is because our current implementation of the network is simply two QPs (*Queue Pair*, the basic connection data structure used in RDMA) between each pair of machines, leading to significant resource contention. For workloads with random and frequent memory access behaviors, DSM becomes the bottleneck.

Network Backends Comparisons: Figure 7(d) gives a breakdown comparison between RDMA and TCP, for big-data workloads. The 2 × 4 setting of GiantVM is used. For all workloads, especially the memory-intensive PR, the RDMA speedup is significant. For PR, 60.1% of DSM time is saved after switching to RDMA, and the total execution time speedup is 1.15×, 1.04×, and 4.46× for WC, Pi, and PR, respectively.

We also evaluate network backends in the lower level of abstraction, in a 2 × 8 GiantVM. Figure 8 reports the latency and bandwidth of three network backends in the host kernel, and the latency of non-owner writes of DSM as CDF.

(1) We measure the latency and bandwidth of three network backends in the *host* kernel, by sending and receiving requests from 64 Byte to 1 MiB for 4,000 times, with a single-thread client and server. Figure 8(a, b) shows TCP-Ethernet > IPoIB > RDMA results for latency and the opposite for bandwidth. IPoIB does not make full use of HCA's capabilities, since the network traffic goes through the TCP/IP stack, and the CPU is not fast enough to process packets for a 56 Gbps IB link. CPU is also a bottleneck for TCP-Ethernet. TCP-Ethernet and IPoIB have a poor bandwidth (<245.7 and 770.0 MiB/s, respectively). While for RDMA, the maximum bandwidth is 2,840.3 MiB/s.

(2) Figure 8(c) shows the 1,000 non-owner write latencies as CDF, while running Sysbench file I/O {1*MiB*,GiantVM} tests. Recall Section 3.3.1, a non-owner write DSM PF incurs the largest overhead among all sorts of DSM PF, thus is the performance bottleneck of DSM. IPoIB has a

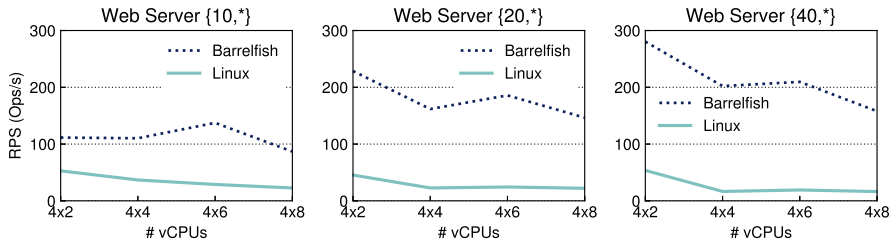

Fig. 9. Optimizing RPS of web servers on GiantVM by running the multi-kernel OS Barrelfish instead of Linux as the guest.

similar performance with RDMA, which shows great stability in latency. While for TCP-Ethernet, the stability in latency is poor. The P90 (90th percentile) latency is 1.8× of P50. For RDMA and IPoIB, this number is 1.25× and 1.48×, respectively. The instability of TCP-Ethernet sometimes leads to failures of timeout-based services when bootstrapping Linux. The respective P90 latencies of RDMA, IPoIB, and TCP-Ethernet are 10.0μs, 18.3μs, and 235.6μs. We run the `Sysbench` local memory write test, and get the P90 latency of 4 KiB DRAM writes to be 0.56μs. RDMA can be a large speedup for DSM, although it is still 17.9× of DRAM.

Summary: GiantVM presents linear scalability for distributed vCPU, and incurs negligible overhead for interrupt and I/O forwarding. DSM component is the performance bottleneck of GiantVM and its overhead increases as the number of vCPUs increases. RDMA brings substantial speedup for DSM, which is a new opportunity for SSI systems.

5.2 DSM-aware Optimization Results

This section answers Question-2 by examining two DSM-aware optimizations. We have a comparison with **Message Passing Interface** (**MPI**) [1], a distributed computation framework, to show the overhead brought by GiantVM (with DSM-aware optimizations enabled) for the *ease* of programming.

Kernel Scalability is Crucial: One significant advantage of a distributed hypervisor over other SSI systems is providing a unified ISA interface, which enables us to utilize many other alternative OSes. Barrelfish is a multi-kernel OS assuming that the cost of cache coherence is expensive and kernel scalability is important. This kind of hardware is more similar to a network or distributed system than a shared-memory architecture on a small-scale machine. There are multiple per-core kernels (hence the multi-kernel) in Barrelfish. The message transmissions among kernels and applications are acted via RPC channels, although the channels are based on the shared memory. It is no doubt that the targeted hardware of Barrelfish perfectly matches the physical reality of GiantVM and minimizes DSM overhead.

We deploy two web servers on Linux (`Apache2`, v2.4.18) and Barrelfish (built-in, commit `78cf89d`) respectively, and use Apache HTTP server benchmarking tool `ab` [17] v2.3 to continuously `GET` a 298 KiB file. The **Request Per Second** (**RPS**) is shown by Figure 9. $\{n, os\}$ means n clients send requests to os simultaneously. Barrelfish outperforms Linux by 6.4× on average. When the concurrency level (the number of concurrent clients) increases, it is clear that Linux lacks scalability and has a large overhead. As a kernel-intensive application, the `Apache2` web server can cause frequent false sharing when accessing shared data structures allocated on the same guest page. Nevertheless, page faults in Barrelfish are limited to the RPC channels, most of which are transmissions of valuable data instead of false sharing. Hence the performance enhancement is determined.

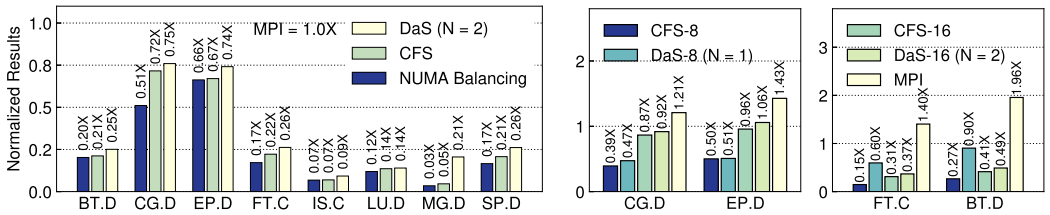

Fig. 10. Results of NPB suites running in DaS. In the left part, 2×8 MPI results are normalized to 1. In the right part, single-node 16-vCPU OpenMP results are normalized to 1.

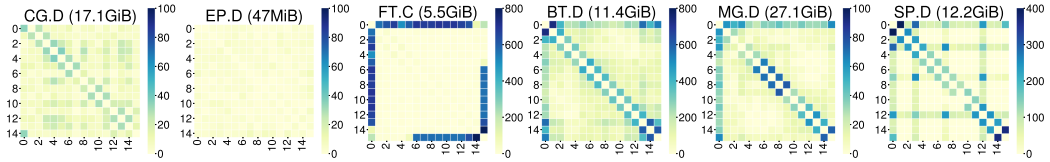

Fig. 11. DSM matrices of representative NPB benchmarks. The frequency of two threads sharing the same guest pages is shown.

However, the single-node web server (8-vCPUs, 40 clients) still has an 8.2× RPS compared to the fastest Barrelfish result (4×2, {40,Barrelfish}), due to the memory intensive nature of web server applications. However, the 6.4× improvement brought by Barrelfish still indicates the importance of kernel scalability in the distributed environment.

DSM-aware Scheduling Results: We use the `NPB` suite v3.4.1 [9] to test the effectiveness of DaS (Section 3.3.2). We choose the OpenMP implementation of `NPB` to run on GiantVM, representing single-machine applications that could benefit from GiantVM. The opposite is the MPI [1] implementation, which adopts a message-passing programming model. We compare the OpenMP version running in DaS and the MPI version, for the "ease of programming" cost.

All tests run on a 2×8 GiantVM, and all benchmarks run 16 threads. Each OpenMP version benchmark runs as a single process. Each benchmark is named as *name.class* where *class* can be `S`, `W`, and `A` – `F`, representing the compiled default input size. We choose *class* as in Figure 10, for the **largest** working sets that could fit into a 64 GiB-RAM GiantVM guest. Since our system does not support compiling class `D` and `E` for `IS`, we use the longest-running `IS.C`.

(1) We first give a comparison between DaS, CFS, and NUMA balancing, in Figure 10 (left-one subfigure). We set $N = 2$ in Algorithm 1 for this experiment, equal to the number of NUMA nodes. The results of 2×8 MPI results are normalized to 1, in which the MPI version of `NPB` runs on two nodes, each node hosts 8 single-threaded processes of the benchmark. OpenMP results are from a single benchmark running on the 16 vCPUs of GiantVM. The total throughput of all 16 threads of benchmarks are compared (Mops/total). The DaS performs the best among the three scheduling policies, which is an increase of up to 3.5× for `MG.D`, compared to CFS, while NUMA balancing leads to a performance collapse of up to 28.7% for `CG.D`. However, the performance gap between MPI and OpenMP continues to exist. Although running a shared address space across two nodes incurs great overhead, DaS could improve the performance by up to 3.5×.

We observe that workloads have variable reactions to different scheduling policies. We analyze it by dumping the DSM matrices while applications are running. Figure 11 gives the most representative ones, with the working set size marked in the title. `EP.D` is insensitive to scheduling policies, since it has infrequent memory accesses. `MG.D` and `SP.D` have relatively regular memory access patterns, and the clique generation algorithm works well on them. For other workloads

with too frequent/infrequent memory accesses, DaS gives a small improvement. For NUMA balancing, it relies on page migration to make all memory accesses local, which can be costly on GiantVM. By examining `/proc/vmstat` file, we observe that NUMA balancing keeps migrating pages between hosts (around 5,000 pages/s). It lacks a history of memory access behaviors of the whole application, introducing large amounts of unnecessary page migration. DaS addresses this issue by memorizing DSM PF statistics in the DSM matrix, during the 6,000 *ms* scheduling period (Section 4.2.2).

(2) CPU and memory are of unequal importance for workloads. Figure 10 (right-two subfigures) reports the comparisons among several configurations for CPU and memory. In this experiment, we run two groups of benchmarks, `CG.D`, `EP.D` for the first group, and `FT.C`, `BT.D` for the second group. Benchmarks within each group achieve comparable run time. The configurations are read as *Scheduler-nCPU*. For $nCPU = 8$, two workloads of the same group run together, scheduled by *Scheduler*. For $nCPU = 16$, a single workload runs in *Scheduler*. All tests except MPI run on a 2×8 GiantVM. For MPI, a single MPI task runs across 2 nodes, each node hosting 8 single-threaded processes for the task.

Note that for DaS-8, we set $N = 1$, thus DaS would schedule all threads of a single process to the same node, eliminating DSM overheads. For DaS-16, $N = 2$. Results of single-node 16-vCPU OpenMP running on the vanilla QEMU-KVM are normalized to 1. *Providing more CPUs* (CFS-16) is more important than *making memory accesses local* (DaS-8) for `CG.D` and `EP.D`, since they do not perform intensive memory accesses (Figure 11). As a result, we see DaS-8 > CFS-16 for them. Also, DaS-16 could achieve up to 1.06× of the single-node OpenMP results, since threads frequently access shared data are scheduled to closer CPU cores, resulting in better use of the L3 cache.

While for `FT.C` and `BT.D`, the results are the opposite. They are the most memory intensive in Figure 11, and DaS could not break their peerthreads into two groups for better memory locality (DaS-16). *Making all memory accesses local* (DaS-8) brings a huge performance boost for them. These workloads are not suitable to run across nodes. For MPI results, due to the total elimination of false sharing by the message-passing model, applications outperform the single-node OpenMP setting, by up to 1.96×. However, they trade ease of programming for performance.

(3) We have an analysis for the overhead that DaS itself introduces. Since the DaS runs kernel threads t_s and t_m on vCPU 0 together with application threads, it incurs CPU and memory overhead. For CPU overhead, we annotate the 6,000 *ms* scheduling period with `getnstimeofday` calls, and collect the time spent running our Algorithm 1, which is at most 79,007 μs. Thus, threads t_s and t_m occupy around 0.16% CPU time, which is negligible. For memory consumption, the main data structure consuming memory is the DSM matrix. Modern applications could have no more than thousands of threads, and the resulting DSM matrix size is 3.8 MiB. As for our experiments with less than 32 threads, a DSM matrix consumes no more than 4 KiB of RAM.

Summary: DaS on Linux guests could offset the cost of programming ease for workloads with moderate memory access frequency (`EP.C`), and improve the performance of memory-intensive workloads by up to 3.5× (`MG.D`). The multi-kernel guest OS could further speed up a memory-intensive web server by 6.4×.

5.3 Benefits of Migration with GiantVM

We first give a comprehensive comparison between migration with GiantVM and VM live migration, in terms of performance degradation and network bandwidth consumption. Then we co-locate LC tasks running on the bare-metal, with migratable BE tasks running on GiantVM, to see the benefits of resource reallocation with GiantVM, including the improved latency of LC tasks and CPU utilization of the cluster. Finally, we answer Question-3 with the experimental results.

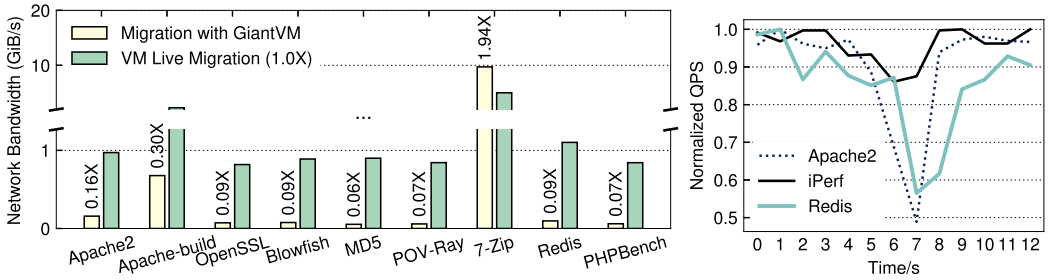

Fig. 12. Network bandwidth comparison between migration with GiantVM and VM live migration (left). Results of VM live migration are normalized to 1. QPS degradation due to migration by GiantVM (right).

Impact of Migration on QPS: In this experiment, we measure the impact on the migrated tasks by GiantVM. We use the 2×8 GiantVM for the experiment, and run `Apache2` v2.4.18, `iPerf` v2.0.5, and `Redis` v3.0.6 on GiantVM. We measure their **Query Per Second** (**QPS**) during migration, i.e. being scheduled in the guest OS from NUMA node 0 to 1. Figure 12 (right) reports the results. The QPS of three benchmarks suffers from a degradation up to 50% in 1–2 s after the migration. Nevertheless, this is not a primary concern for a BE task running on GiantVM.

For VM live migration, these applications cannot respond to the requests during downtime, i.e. the time during which applications are stopped and the guest pages are transferred, and the QPS would be 0. The downtime comparison is omitted here because the concept is confusing for migration with GiantVM.

Network Bandwidth Consumption of Migration: Our main claim for the benefit of migration with GiantVM is the low cost of migration, i.e., the network bandwidth consumption is small, compared to VM live migration. Note that in our evaluation, the destination has been scheduled at least **once** for GiantVM, to allow standby pages at the migration destination. The comparisons with VM live migration are carried out with a 2×8 GiantVM. While for VM live migration configurations, we use the default settings of QEMU v2.8.1.1, in which `Postcopy-RAM` and `XBZRLE` [58] are enabled. The migrated workloads are `Apache2` v2.4.18, `Apache-build` (build the `Apache2` v2.4.18 from source code), `OpenSSL` v1.0.2, `Blowfish` v2.2.2 [29], `MD5` [64], `POV-Ray` v3.7, `7-Zip` v19.00, `Redis` v3.0.6, and `PHPBench` v1.1.0.

Figure 12 (left) presents the average network bandwidth consumption during one migration. GiantVM can save 68% bandwidth on average (the median is 91%). The cost of migration with GiantVM heavily depends on the fraction of read-only pages of a workload (Section 3.3.4). `7-Zip` frequently `malloc/free` bulk memory, which tends to touch a huge range of address space during the migration. Due to the lack of the stop-and-copy stage in migration with GiantVM, pages are frequently transferred across nodes due to DSM PF during migration, and the bandwidth consumption rises to 1.94× of VM live migration. While for most applications, there are still many read-only pages that can be exploited for lowering network cost. VM live migration could not efficiently detect them.

Application Co-location Results with LaS: The effectiveness of LaS (Section 4.2.3) is examined in this part. We have three settings for comparison, without co-location (*w/o*), naïve co-location (*naïve*), and LaS co-location (*LaS*). The tasks without co-location achieve the maximum performance but leave a high surplus utilization. The naïve co-location only co-locates fixed tasks, which lacks the ability of resource reallocation. So it suffers from performance degradation and QoS violation. LaS co-location runs LC tasks as fixed tasks on the bare-metal, and BE tasks as migratable tasks on GiantVM. When the CPU requirement of LC tasks bursts, LaS schedules BE tasks

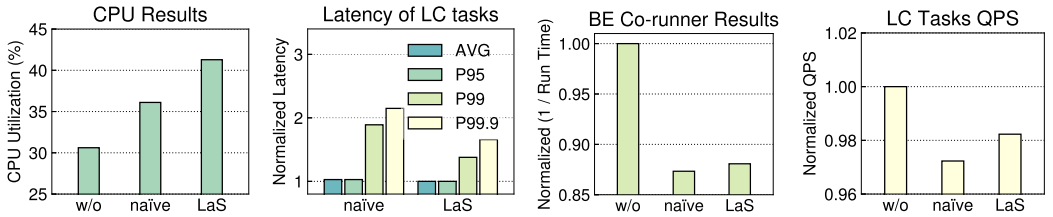

Fig. 13. Comparisons of CPU utilization results, LC tasks latencies, BE tasks run time, and LC tasks QPS, among three co-location settings, which are naïve co-location (naïve), LaS co-location (LaS), and without co-location (w/o).

away from the overloaded node, to ensure QoS of LC tasks. Meanwhile, BE tasks could make use of the surplus CPU resources on nodes with a low `steal time` (Section 4.2.3).

In practical use, LC tasks are interactive jobs, such as a web server and an in-memory key-value store. They have a strict demand for low latency and high QoS. BE tasks are batch jobs like data processing and kernel compiling. BE tasks can be sent to run on GiantVM, without any modification. We run `OpenSSL` v1.0.2 as BE tasks on a 4×8 GiantVM, and `Memcached` v1.6.12 as LC tasks on the first physical node hosting GiantVM (Node 0). For *w/o*, we run `Memcached` alone on Node 0, then run `OpenSSL` on an 8-vCPU QEMU alone on Node 0. For *naïve*, we run `Memcached` and 8-vCPU QEMU hosting `OpenSSL` together on Node 0. For `Memcached`, latencies and QPS are measured. For `OpenSSL`, the reciprocal of the run time is measured. The `Memcached` workload follows a Gaussian distribution.

Figure 13 shows the results. LaS decreases the P95, P99, and P99.9 latency of `Memcached` by 26.7%, 27.1%, and 22.8%, respectively, compared to the naïve co-location. Also, CPU utilization is improved by 14.3%, compared to the naïve co-location. For QPS and run time reciprocal results, we could see w/o > LaS > naïve. Hence, the extra 14.3% CPU utilization can be exploited by LaS to provide a better QoS guarantee for both BE and LC tasks.

Summary: As a novel approach for workload migration, GiantVM could save 68% bandwidth on average compared to VM live migration, and no downtime is introduced. Resource reallocation with GiantVM could improve CPU utilization by 14.3%, compared to the naïve co-location. As measured in Section 5.1, interrupt (including IPI) forwarding overhead is negligible in GiantVM, thus distributed vCPU could be very scalable. GiantVM as a job scheduler could be deployed on dozens of physical nodes if all workloads on GiantVM are scheduled to a single NUMA node.

6 DISCUSSION

6.1 Memory Consistency Model

Currently, GiantVM supports a strict memory model, the sequential consistency. Generally speaking, relaxing the memory model can enhance performance. On the x86 platforms, the memory model is x86-TSO [55], 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`[10] instruction flushes the store buffer, and buffered write can be broadcasted at any time.

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

We discuss the possibility of implementing 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 adding a hypercall after every `mfence`[11] of the guest. This is impractical since `mfence` is a user-level instruction and is used everywhere. For example, `mfence` is generated for operations on Java `volatile` keywords, as well as C++ `atomic` variables. 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 [6]. 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 *diff*s.

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 unaware 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 assumption that the underlying physical machines/VMs are unreliable.

7 RELATED WORK

Single System Image: Rajkumar Buyya et al. conducted comprehensive surveys of SSI before 2016 [33]. The early SSI systems include distributed OSes or cluster OSes. They explore many interesting technologies like process migration, process checkpoints, shared file systems, and so on. Since the 2,000s, it tends to be extended based on existing OSs such as Linux to have SSI features, including Kerrighed [60] and MOSIX [10]. MOSIX provides the process migration capability but is unable to deal with the residual dependency problem, while GiantVM could solve it. Recent distributed OSes include Helios [49], Popcorn [11], and LegoOS [65]. A shortcoming of distributed OS is that it lacks drivers of heterogeneous devices such as FPGA and GPU, which are not open-source and are impractical to port to a dedicated OS. GiantVM enjoys the ease of programming of SSI and the device drivers on Linux in the meantime.

Recently, there has been an interest in connecting physical machines with high-speed networks to construct SSI. FireBox [7] and The Machine [35] provide a single machine composed of numerous computing sockets and DRAMs. Memory Blade [42], Infiniswap [32], and Remote Regions [4] enable local applications to access memory on the remote node. ReFlex [38] and vSAN [30] 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

[11] 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.

[3] since the debut of Intel VT-x [59]. In addition to KVM, Xen [12] 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. VirtIO [52] uses a para-virtualization method, which lets the guest OS interact with virtualized devices directly. SR-IOV [28] virtualizes one physical device into multiple instances. It resembles the distributed I/O design of GiantVM, i.e., multiplexing the device on the node of the master QEMU for other QEMU instances of GiantVM. Although SR-IOV could lower hardware costs while allowing direct access from virtual machines, the GiantVM I/O, and interrupt forwarding mechanism enables I/O load balancing by assigning devices to separate nodes, without specialized hardware support.

Software and Hardware-based DSM: Protic et al. [51] give a good survey of software-based DSM researches in the 1990s. Ivy [41] is the ancestor of most modern DSM systems. Mirage [31] improves Ivy by guaranteeing page ownership for a fixed period to mitigate page thrashing. Munin [19] provides different pages with different coherence mechanisms. TreadMarks [6] uses *diff* algorithms to reduce network transfers. Hotpot [56], Grappa [48], and GAM [18] are the recent work of integrating DSM and RDMA.

Hardware-based DSM includes Scale-out NUMA [50] and Venice [27]. Scale-out NUMA directly layers RDMA-inspired programming model on a NUMA memory fabric, targeted at fine-grained remote memory accesses. This facilitates an efficient software-based DSM for GiantVM, which we will explore in the future. Venice eliminates the need to port to a new programming model and provides a strong communication substrate and resource-joining mechanisms allowing user applications to efficiently leverage non-local resources.

SSI by Virtualization: ScaleMP [54] proposes the *vSMP Foundation*, a *type-I* hypervisor with a 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 that 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 cache line definition of the Linux kernel to 4 KB. TidalScale supports similar features of ScaleMP. Its highlight is the support of the migration of vCPUs across the nodes to mitigate page thrashing [14].

vNUMA [20] is a *type-I* distributed hypervisor on Itanium leveraged by pre-virtualization [40]. Its main contribution is the enhancement of the Ivy protocol for underlying DSM. Nevertheless, 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. 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 AND FUTURE WORK

In this article, we present GiantVM, a distributed hypervisor for the SSI abstraction based on distributed shared memory, implementing a 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. It 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 to apply 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. Also, GiantVM enables an efficient and economical approach to workload migration. In the future, we could remove the performance bottleneck by relaxing the memory model implemented by DSM to be x86-TSO or applying new hardware to replace our software DSM.

ACKNOWLEDGEMENTS

We sincerely thank David Kaeli and the anonymous reviewers for their feedback. We also sincerely thank Yufan Jiang, Zhengjun Zhang, Bo Peng, and Yanqiang Liu for their invaluable help.

REFERENCES

[1] The MPI Forum. 1993. CORPORATE the MPI forum - MPI: A message passing interface. In *Proceedings of the Supercomputing'93, Portland, Oregon, USA, November 15-19, 1993*, Bob Borchers and Dona Crawford (Eds.), ACM, 878–883. DOI: https://doi.org/10.1145/169627.169855

[2] 2014. *THE DESIGN OF CFS*. Retrieved 12 July, 2021 from https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt.

[3] 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*. 2–13. DOI: https://doi.org/10.1145/1168857.1168860

[4] 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 *Proceedings of the 2018 USENIX Annual Technical Conference*. 775–787. Retrieved from https://www.usenix.org/conference/atc18/presentation/aguilera.

[5] Alibaba. 2021. *Alibaba Production Cluster Trace Data*. Alibaba Retrieved from https://github.com/alibaba/clusterdata.

[6] 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. DOI: https://doi.org/10.1109/2.485843

[7] Krste Asanović. 2014. FireBox: A hardware building block for 2020 warehouse-scale computers. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*.

[8] Infiniband Trade Association. 2008. *InfiniBand Architecture Volume 1, General Specifications*. Infiniband Trade Association.

[9] David H. Bailey, Eric Barszcz, John T. Barton, D. S. Browning, Robert L. Carter, Leonardo Dagum, Rod A. Fatoohi, Paul O. Frederickson, T. A. Lasinski, Robert Schreiber, Horst D. Simon, V. Venkatakrishnan, and Sisira Weeratunga. 1991. The nas parallel benchmarks. *International Journal of High Performance Computing Applications* 5, 3 (1991), 63–73. DOI: https://doi.org/10.1177/109434209100500306

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

[11] 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 10th European Conference on Computer Systems*. ACM, New York, NY, Article 29, 16 pages. DOI: https://doi.org/10.1145/2741948.2741962

[12] 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*. 164–177. DOI: https://doi.org/10.1145/945445.945462

[13] Andrew Baumann, Paul Barham, Pierre-Évariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009*. 29–44. DOI: https://doi.org/10.1145/1629575.1629579

[14] C. Gordon Bell and Ike Nassi. 2018. Revisiting scalable coherent shared memory. *Computer* 51, 1 (2018), 40–49.

[15] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 41–41. Retrieved from http://dl.acm.org/citation.cfm?id=1247360.1247401.

[16] Kamanashis Biswas and Md. Ashraful Islam. 2009. Hardware virtualization support in INTEL, AMD and IBM power processors. arXiv:0909.0099. Retrieved from http://arxiv.org/abs/0909.0099.

[17] Justin F. Brunelle, Michael L. Nelson, Lyudmila Balakireva, Robert Sanderson, and Herbert Van de Sompel. 2013. Evaluating the SiteStory transactional web archive with the ApacheBench tool. In *Proceedings of the International Conference on Theory and Practice of Digital Libraries*. Springer, 204–215.

[18] 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. DOI: https://doi.org/10.14778/3236187.3236209

[19] John B. Carter, John K. Bennett, and Willy Zwaenepoel. 1991. Implementation and performance of munin. In *Proceedings of the 13th ACM Symposium on Operating System Principles*. Henry M. Levy (Ed.), ACM, 152–164. DOI: https://doi.org/10.1145/121132.121159

[20] Matthew Chapman and Gernot Heiser. 2009. vNUMA: A virtual shared-memory multiprocessor. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 2–2. Retrieved from http://dl.acm.org/citation.cfm?id=1855807.1855809.

[21] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-aware resource partitioning for multiple interactive services. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.), ACM, 107–120. DOI: https://doi.org/10.1145/3297858.3304005

[22] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live migration of virtual machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*. Amin Vahdat and David Wetherall (Eds.), USENIX. Retrieved from http://www.usenix.org/events/nsdi05/tech/clark.html.

[23] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert Tappan Morris, and Eddie Kohler. 2013. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the ACM SIGOPS 24th Symposium on Operating Systems Principles*. Michael Kaminsky and Mike Dahlin (Eds.), ACM, 1–17. DOI: https://doi.org/10.1145/2517349.2522712

[24] Russ Cox, M. Frans Kaashoek, and Robert Morris. 2020. Xv6, a Simple Unix-like Teaching Operating System.

[25] Yan Cui, Yingxin Wang, Yu Chen, and Yuanchun Shi. 2013. Lock-contention-aware scheduler: A scalable and energy-efficient method for addressing scalability collapse on multicore systems. *ACM Transactions on Architecture and Code Optimization* 9, 4 (2013), 44:1–44:25. DOI: https://doi.org/10.1145/2400682.2400703

[26] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Communications of the ACM* 51, 1 (2008), 107–113. DOI: https://doi.org/10.1145/1327452.1327492

[27] Jianbo Dong, Rui Hou, Michael C. Huang, Tao Jiang, Boyan Zhao, Sally A. McKee, Haibin Wang, Xiaosong Cui, and Lixin Zhang. 2016. Venice: Exploring server architectures for effective resource sharing. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture*. IEEE Computer Society, 507–518. DOI: https://doi.org/10.1109/HPCA.2016.7446090

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

[29] Artyom Egorov. 2020. Blowfish Encryption Library for Browsers and Node.js. Retrieved 21 Dec., 2020 from https://github.com/egoroof/blowfish.

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

[31] B. Fleisch and G. Popek. 1989. Mirage: A coherent distributed shared memory design. *SIGOPS Operating Systems Review* 23, 5 (1989), 211–223. DOI: https://doi.org/10.1145/74851.74871

[32] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient memory disaggregation with infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*. 649–667. Retrieved from https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu.

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

[34] Yang Hong, Yang Zheng, Fan Yang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Scaling out NUMA-aware applications with RDMA-based distributed shared memory. *Journal of Computer Science and Technology* 34, 1 (2019), 94–112. DOI: https://doi.org/10.1007/s11390-019-1901-4

[35] 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*. 1:1. DOI: https://doi.org/10.1145/2768405.2768406

[36] Colin Ian King. 2017. Stress-ng. Retrieved March 28, 2018 from http://kernel.ubuntu.com/git/cking/stressng.git.

[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*.

[38] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote flash ≈ local flash. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*. 345–359. DOI: https://doi.org/10.1145/3037697.3037732

[39] Alexey Kopytov. 2004. Sysbench: A system performance benchmark. Retrieved 11 August, 2021 from http://sysbench.sourceforge.net/.

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

[41] Kai Li and Paul Hudak. 1989. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems* 7, 4 (1989), 321–359. DOI: https://doi.org/10.1145/75104.75105

[42] 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 *Proceedings of the 36th International Symposium on Computer Architecture*. 267–278. DOI: https://doi.org/10.1145/1555754.1555789

[43] Tongping Liu and Emery D. Berger. 2011. SHERIFF: Precise detection and automatic mitigation of false sharing. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Cristina Videira Lopes and Kathleen Fisher (Eds.), ACM, 3–18. DOI: https://doi.org/10.1145/2048066.2048070

[44] Yoshihiro Mazda, Daijiro Kobashi, and Satoshi Okada. 2005. Tidal-scale hydrodynamics within mangrove swamps. *Wetlands Ecology and Management* 13, 6 (2005), 647–655.

[45] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. 2000. Process migration. *ACM Computing Surveys* 32, 3 (2000), 241–299. DOI: https://doi.org/10.1145/367701.367728

[46] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. 2008. Containers checkpointing and live migration. In *Proceedings of the Ottawa Linux Symposium*.

[47] Shripad Nadgowda, Sahil Suneja, Nilton Bila, and Canturk Isci. 2017. Voyager: Complete container state migration. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems*. Kisung Lee and Ling Liu (Eds.), IEEE Computer Society, 2137–2142. DOI: https://doi.org/10.1109/ICDCS.2017.91

[48] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 USENIX Annual Technical Conference*. 291–305. Retrieved from https://www.usenix.org/conference/atc15/technical-session/presentation/nelson.

[49] 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*. ACM, New York, NY, 221–234. DOI: https://doi.org/10.1145/1629575.1629597

[50] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*. Rajeev Balasubramonian, Al Davis, and Sarita V. Adve (Eds.), ACM, 3–18. DOI: https://doi.org/10.1145/2541940.2541965

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

[52] Rusty Russell. 2008. Virtio: Towards a de-facto standard for virtual I/O devices. *SIGOPS Operating Systems Review* 42, 5 (2008), 95–103. DOI: https://doi.org/10.1145/1400097.1400108

[53] Jerome H. Saltzer, Roy Levin, and David D. Redell (Eds.). 1983. In *Proceedings of the 9th ACM Symposium on Operating System Principles*. ACM. DOI: https://doi.org/10.1145/800217

[54] Dirk Schmidl, Christian Terboven, Andreas Wolf, Dieter an Mey, and Christian H. Bischof. 2010. How to scale nested OpenMP applications on the ScaleMP vSMP architecture. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing*. IEEE Computer Society, 29–37. DOI: https://doi.org/10.1109/CLUSTER.2010.38

[55] 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. *Communications of the ACM* 53, 7 (2010), 89–97. DOI: https://doi.org/10.1145/1785414.1785443

[56] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, New York, NY, 323–337. DOI: https://doi.org/10.1145/3127479.3128610

[57] 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 12th European Conference on Computer Systems*. 1–15. DOI: https://doi.org/10.1145/3064176.3064189

[58] Petter Svärd, Benoit Hudzia, Johan Tordsson, and Erik Elmroth. 2011. Evaluation of delta compression techniques for efficient live migration of large virtual machines. In *Proceedings of the 7th International Conference on Virtual Execution Environments*. Erez Petrank and Doug Lea (Eds.), ACM, 111–120. DOI: https://doi.org/10.1145/1952682.1952698

[59] 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. DOI: https://doi.org/10.1109/MC.2005.163

[60] 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. DOI: https://doi.org/10.1142/S0129626403001185

[61] Rik van Riel and Vinod Chegu. 2014. Automatic NUMA balancing. In *Proceedings of the Red Hat Summit* (2014).

[62] Xiaoyuan Wang, Haikun Liu, Xiaofei Liao, Ji Chen, Hai Jin, Yu Zhang, Long Zheng, Bingsheng He, and Song Jiang. 2019. Supporting superpages and lightweight page migration in hybrid memory systems. *ACM Transactions on Architecture and Code Optimization* 16, 2 (2019), 11:1–11:26. DOI: https://doi.org/10.1145/3310133

[63] Charles F. Webb. 2008. IBM z10: The next-generation mainframe microprocessor. *IEEE Micro* 28, 2 (2008), 19–29. DOI: https://doi.org/10.1109/MM.2008.26

[64] Bryce Wilson. 2021. MD5 in C. Retrieved 21 Dec., 2020 from https://github.com/Zunawe/md5-c.git.

[65] Yilun Chen Yizhou Shan, Yutong Huang, and Yiying Zhang. 2018. Lego: A decomposed, distributed OS for hardware resource disaggregation. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, Carlsbad, CA. Retrieved from https://www.usenix.org/conference/osdi18/presentation/shan.

[66] 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*. USENIX Association, Berkeley, CA, 2–2. Retrieved from http://dl.acm.org/citation.cfm?id=2228298.2228301.

[67] Jin Zhang, Zhuocheng Ding, Yubin Chen, Xingguo Jia, Boshi Yu, Zhengwei Qi, and Haibing Guan. 2020. GiantVM: A type-II hypervisor implementing many-to-one virtualization. In *VEE'20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, virtual event*. Santosh Nagarakatte, Andrew Baumann, and Baris Kasikci (Eds.), ACM, 30–44. DOI: https://doi.org/10.1145/3381052.3381324

[68] 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*. ACM, New York, NY, 523–536. DOI: https://doi.org/10.1145/2785956.2787484

[69] 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. DOI: 10.1109/MC.2005.163

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

Received August 2021; revised November 2021; accepted December 2021