High-density Multi-tenant Bare-metal Cloud

Xiantao Zhang
Alibaba Group
xiantao.zxt@alibaba-inc.com

Xiao Zheng
Alibaba Group
zhengxiao.zx@alibaba-inc.com

Zhi Wang
Florida State University
zwang@cs.fsu.edu

Hang Yang
Alibaba Group
yanghang.yh@alibaba-inc.com

Yibin Shen
Alibaba Group
zituan@taobao.com

Xin Long
Alibaba Group
longxin.xl@alibaba-inc.com

Abstract

Virtualization is the cornerstone of the infrastructure-as-a-service (IaaS) cloud, where VMs from multiple tenants share a single physical server. This increases the utilization of data-center servers, allowing cloud providers to provide cost-efficient services. However, the multi-tenant nature of this service leads to serious security concerns, especially in regard to side-channel attacks. In addition, virtualization incurs non-negligible overhead in the performance of CPU, memory, and I/O. To this end, the bare-metal cloud has become an emerging type of service in the public clouds, where a cloud user can rent dedicated physical servers. The bare-metal cloud provides users with strong isolation, full and direct access to the hardware, and more predicable performance. However, the existing single-tenant bare-metal service has poor scalability, low cost efficiency, and weak adaptability because it can only lease *entire physical servers* to users and have no control over user programs after the server is leased.

In this paper, we propose the design of a new high-density multi-tenant bare-metal cloud called BM-Hive. In BM-Hive, each bare-metal guest runs on its own compute board, a PCIe extension board with the dedicated CPU and memory modules. Moreover, BM-Hive features a hardware-software hybrid virtio I/O system that enables the guest to directly access the cloud network and storage services. BM-Hive can significantly improve the cost efficiency of the bare-metal service by hosting up to 16 bare-metal guests in a single physical server. In addition, BM-Hive strictly isolates the bare-metal guests at the hardware level for better security and isolation. We have deployed BM-Hive in one of the largest public cloud infrastructures. It currently serves tens of thousands of users at the same time. Our evaluation of BM-Hive demonstrates its strong performance over VMs.

CCS Concepts. • **Security and privacy** → **Virtualization and security**; • **Computer systems organization** → **Cloud computing**.

Keywords. virtualization, bare-metal cloud, High-density, cloud infrastructure

ACM Reference Format:
Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. 2020. High-density Multi-tenant Bare-metal Cloud. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3373376.3378507

1 Introduction

Virtualization is the cornerstone of the public IaaS cloud. It enables the cloud provider to multiplex VMs from several tenants onto the same physical server. Recent advancement in the virtualization technology, such as Extended Page Table (EPT), IOMMU, and para-virtualized I/O, has significantly improved the performance of the VM, making the cloud a viable and cost-effective choice for many computation tasks. However, the multi-tenant nature of the VM-based cloud has led to substantial concerns in security and performance – VMs from different users share the same physical server. The interference in the shared resources can be exploited for side-channel attacks, where the attacker infers important information (e.g., the encryption key) by observing the victim's resource usage patterns[24, 27, 35]. This is particularly problematic for tightly-coupled resources, such as the CPU cache and hyper-threading. Conflicts in the shared resources can also lead to unpredictable performance fluctuations [16], on top of the existing overhead of virtualization, leaving cloud users that demand high CPU and memory performance unsatisfied. As such, the VM-based cloud falls short in the security, isolation, and performance for more demanding cloud services such as 3D rendering, gaming, and high-frequency stock trading.

Limitations of existing approach: to address these concerns, a new type of the cloud services has emerged, the bare-metal service, where a cloud user can rent dedicated

Service	Security	Isolation	Performance	Density
VM-based cloud	Side-channel and DoS attacks because of resource sharing	Weak isolation because of resource sharing	CPU, memory, and I/O overhead caused by virtualization	Very high density through server over-provisioning
Single-tenant bare-metal cloud	Poor security because the untrusted user has unfettered access to the whole platform	Strong isolation due to exclusive access to the system (a moot point because of poor security)	Native performance	Very low density, one user per server, leading to high cost
BM-Hive	No side-channel or DoS attacks due to hardware-based isolation; Protected hardware resources, particularly the firmware	Strong hardware-based isolation	Native CPU and memory performance; para-virtualized I/O with minor overhead	High, 16 bm-guests per server at most

Table 1. Comparison of three cloud services

physical servers. We call this service as single-tenant bare-metal server because such a server can only be rented, in its entirety, to one user at a time. Therefore, the user has exclusive, full access to the hardware and the complete freedom to run the OS of choice. This provides the user the same level of security and performance isolation as the physical server, as well as the elasticity of the cloud. However, this approach has severe limitations in cost-efficiency, performance, and Interoperability:

- **Cost:** single-tenant bare-metal service can only lend whole physical servers, which often have higher configurations than what most users need. For example, more than 95% of the VMs in our cloud use less than 32 CPU cores (we expect the bare-metal service to be similar), while most cloud servers have more than 64 CPU cores, with a maximum of 128 cores. This makes the single-tenant bare-metal service much less cost efficient.
- **Single-thread performance:** most cloud servers feature Intel Xeon processors. High-core count Xeon processors, commonly used in the cloud infrastructure, have a relatively low base clock speed (2.xGHz). For example, the single-thread performance of Core i7-8086K is 1.6x of that of Xeon E5-2699v4 in the CPU Mark [8]. Users of the bare-metal service often require better single-thread performance that is only available in desktop or low-end Xeon processors.
- **Interoperability and manageability:** the bare-metal service must be seamlessly integrated into the existing cloud system in order to benefit from the elasticity of the cloud infrastructure. However, the design of single-tenant bare-metal servers are significantly different from the VM-based cloud. They are often managed by the out-of-band management system like Intel Management Engine, which is known to be insecure [3]. In addition, the bare-metal service should be compatible with the VM image so that the user can utilize the same image for both VM-based and bare-metal services.

Because of those limitations, single-tenant bare-metal service is not the right approach for the bare-metal cloud service. There is a pressing need for the real bare-metal cloud that is secure for both the cloud provider and users alike, cost-efficient, featuring strong single-thread performance, and seamlessly integrated into the cloud infrastructure.

Our approach: In this paper, we present the design of BM-Hive, an extensible multi-tenant bare-metal service, that can address all these challenges. An BM-Hive server consists of three components: a bare-metal hypervisor, an array of bare-metal guests, and IO-Bond. For brevity, we call the bare-metal hypervisor and guest as bm-hypervisor and bm-guest, respectively, and call their VM-based counterparts vm-hypervisor and vm-guest. The bm-hypervisor runs on the main CPU of the server. It manages the life cycle of all its bm-guests (power on/off etc) and acts as the back-end for IO-Bond. Unlike the vm-hypervisor, the bm-hypervisor does not need to virtualize the CPU or the memory. Instead, each bm-guest runs on its own compute board, which consists of the dedicated processor and memory units and the PCIe interface for I/O. IO-Bond is a piece of hardware, implemented currently by FPGA, that connects a compute board to the main board. On one hand, it emulates a number of I/O devices (e.g., virtio network and block devices) on the PCIe bus of the compute board; on the other hand, it connects the compute board to the main board's PCIe bus and forwards the I/O requests from the bm-guest to the bm-hypervisor for processing. Consequently, the design of main board can be simplified as possible, as long as it covers the need of bm-hypervisor and I/O processing, while the compute board can support any processors, including Xeon and high-frequency Core i7 and i9 processors, as long as it follows the same I/O interface. Particularly, the CPU choice of the compute board is independent from that of the main board.

The design of BM-Hive adequately addresses all the limitations of the single-tenant bare-metal service. **First**, BM-Hive can support at most 16 bm-guests on a single BM-Hive server. This significantly improves the density of the service and lowers the cost because the user only needs to rent the most suitable bm-guest. **Second**, a bm-guest can utilize any processor supported by the compute board, including desktop processors with high single-thread performance. **Lastly**, BM-Hive can be seamlessly integrated into the existing cloud infrastructure. The main difference from the VM-base cloud service is that vm-guests run on virtual CPUs, while bm-guests run on their own compute boards. This difference is transparent to the cloud infrastructure, thanks to the bm-hypervisor and IO-Bond. Table 1 compares the different aspects of the three types of the cloud service. Besides, a bm-guest does not have unfettered control over the whole server. The bm-hypervisor controls its execution via the PCIe interface. The firmware of the compute board is properly signed, and can only be updated if the signature of the new firmware passes the verification. In addition, IO-Bond enforce the proper isolation of bm-guests in the hardware

We have deployed BM-Hive in our data centers that serve one of the largest public IaaS clouds. It's currently serving tens of thousands of users each day. We thoroughly evaluated the security and performance of BM-Hive. Our experiments show that BM-Hive has strong isolation, security, and performance. Particularly, its performance of the commonly deployed applications such as NGINX and MariaDB is substantially better than that of the VM. For example, it is 50% faster for NGINX than a similarly equipped vm-guest (with lower cost).

In summary, this paper makes the following contribution:

- We propose a new design for the bare-metal cloud that can combine the advantages of existing VM-based and single-tenant bare-metal server, and provide secure, flexible, and interoperable bare-metal cloud service.
- BM-Hive features a hardware-software hybrid virtio system that allows the bare-metal guests to be directly integrated into the existing cloud infrastructure, immediately gaining the availability and elasticity of the cloud service.
- We have built a product-grade system based on the design of BM-Hive and deployed it in the real cloud for public use. This ultimately demonstrates the feasibility of BM-Hive.
- We have deployed BM-Hive in one of the largest public cloud infrastructures and thoroughly measured its performance. In particular, we demonstrated its advantages over the VM-based cloud service.

The rest of the paper is organized as follows. We first further motivate the problem by quantifying the impacts of virtualization on the vm-guest performance in the real cloud environment in Section 2. We then present the design and evaluation of BM-Hive in Section 3 and 4, respectively. Section 5 compares BM-Hive to the related work, and Section 6 discusses the potential improvements to BM-Hive. We conclude the paper in Section 7.

2 Background and Motivation

In this section, we further motivate the problem by measuring the overhead of virtualization in the real cloud. These non-negligible overhead make us to develop BM-Hive for our users.

Background: virtualization is the key enabling technology for the traditional cloud because of its flexibility and manageability. For example, a VM encapsulates all the current (virtual) hardware and software states of the guest operating system; it can be migrated from one physical server to another while the guest is running, i.e., the so-called VM live-migration [15]. In addition, the hypervisor has full control over the guests. It can thus enforce isolation among them and protect itself and the system from malicious guests.

Three major aspects of virtualization are the virtualization of CPU, memory, and I/O. CPU and memory are virtualized directly by the virtualization extension of the processor. For example, Intel VT adds a more-privileged root mode for the hypervisor and runs guests in the non-root mode. The root mode can monitor the guest execution and trap virtualization-related events (e.g., updates to the control registers). The memory is virtualized by two-level paging (e.g., Intel's EPT and AMD's NPT) where a second-level page table is added to map the guest physical memory to the actual physical memory. Nowadays, the I/O in the cloud is mostly supported by para-virtualization, where drivers in the guest kernel directly request and serve high-level I/O operations, instead of emulating low-level I/O operations such as register access. Guests may also directly access hardware devices through device pass-through.

2.1 Impact on Performance

Virtualization's impact on performance is twofold: virtualizing system resources introduces non-trivial performance overhead, and the interference in the shared resources leads to performance variation.

Performance overhead: the overhead of virtualization directly correlates with VM exits, where the CPU exits from the guest mode and returns to the hypervisor to handle special events. Many events in the guest can cause VM exits, such as updates to MSRs (model-specific register), IPIs (inter-processor interrupts), and certain page faults. These events have to be handled by the hypervisor to safely virtualize system resources and enforce proper isolation. It takes about 10 μs for the KVM hypervisor [1] to handle an event, but could be longer if the event handler is preempted by the kernel. The performance overhead becomes observable when there are more than 5,000 VM exits per second. We conducted a quick count of VM exits on 300,000 VMs in our cloud data center for five minutes. The result is shown in Table. 2. For example, there were 1,110 VMs having more than 50,000 VM exits per second for each virtual CPU (vCPU) during this period, i.e., about 50% of the CPU time is spent in VM exits.

# of VM exits	Percent of VMs
10K	3.82%
50K	0.37%
100K	0.13%

Table 2. Number of VM exits per second per vCPU

[1]Our hypervisor is based on KVM.

Device virtualization can also cause overhead even with device pass-through which grants a VM direct access to the hardware device. Device pass-through is often deployed in the cloud to assign network adapters and GPUs to VMs. It relies on IOMMU to confine the DMA access from the assigned devices. The use of IOMMU can increase the CPU usage by as much as 60% for network adapters, even though the network throughput remains mostly the same [14]. The lost CPU time should have been used by the guest computation.

There are many other aspects of virtualization that contributes to performance overhead, such as the lock holder preemption (a vCPU is preempted while holding a lock), TLB shooting-down, TLB missing, etc. None of these issues, including VM exits and IOMMU slowdown, exist in the BM-Hive server.

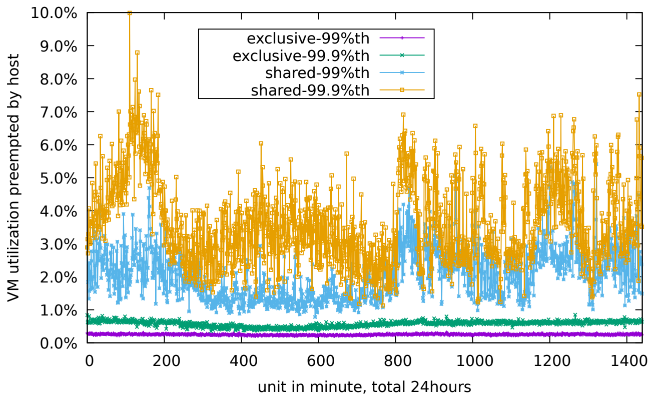

Figure 1. VM preemption of the 99th and 99.9th percentile. The y-axis shows the percent of the CPU time used by the hypervisor/the host OS.

Performance variation: the VM's performance can be affected by conflicts in the shared resources in varying ways. For example, on a busy server, it could take the full load of 8 to 10 CPU cores for the hypervisor to serve I/Os and other requests from the VMs. The tasks of the hypervisor and the host OS can preempt the execution of the guest VMs, causing both performance overhead and variation.

To quantify the impact of VM preemption, we recorded the execution events of $20,000$ VMs in our datacenter for 24 hours, and ordered them from low to high according to the CPU preemption rate by the hypervisor. The result is reported in Fig. 1. The VM preemption rate is the percent of the VM's lifetime when the VM's vCPUs are preempted by the host tasks. The figure reports the data for 99th and 99.9th percentiles. The 99th percentile data shows the VM preemption rate of the $19,800$th VM ($20,000 \times 99\%$) in the ordered list, i.e., there were 1% of these VMs perform worse than it. The 99.9th percentile is similarly defined. These two percentiles are an important performance metric. Moreover, the shared in the figure means that the vCPUs of the VM are not pinned to the physical cores; the exclusive means the opposite. Normally, high-end VMs are pinned for better performance and minimal variation. The figure shows that the 99th percentile of the shareable VMs were preempted by the host from about 2% to 4%, and the 99.9th percentile of the shareable VMs were preempted from 2% to 10%. The situation for the exclusive VMs is both better (about 0.2% and 0.5%, respectively) and more stable. Such variation and preemption, albeit relatively small, can cause real problems for demanding services, such as high-frequency stock trading and game streaming.

Note that in these cases none of the co-resident VMs are actively trying to monopolize system resources. The situation will be significantly worse otherwise. For example, a malicious VM can substantially slow-down other co-resident VMs by repeatedly flushing the shared (L3) CPU cache with its own data.

2.2 Security Concerns

Virtualization provides sufficient isolation between VMs. However, such isolation is enforced mostly by the software. Hardware resources are still shared. Resource sharing is the leading cause of concern for side-channel attacks, especially after the recent discovery of a series of flaws in the CPU's speculative execution and caches (i.e., Meltdown, Spectre, L1TF, etc) [22, 24, 26]. These flaws can be exploited by side-channel attacks to steal sensitive data. They have become a serious concern for sensitive users. In BM-Hive, bm-guests are physically isolated; side-channel attacks are thus not a concern.

Vulnerabilities in the hypervisor are another security concern in the cloud. Linux/KVM is probably the most popular hypervisor used by public clouds. They are highly complex and contain many known and unknown vulnerabilities – there are 170 CVEs (Common Vulnerabilities and Exposures) reported for the Linux kernel and KVM in 2018 alone [10]. The hypervisor is the main attack surface in the cloud. For example, the instruction emulation of KVM is one of the most vulnerable components of KVM, and it is directly exposed to the (malicious) guests. Compared to the vm-hypervisor, bm-hypervisor is much simpler because it does not need CPU and memory virtualization (the instruction emulation is a part of the CPU virtualization); and it is not directly accessible to the guests.

2.3 Nested Hypervisor

Traditional cloud is a poor choice for users who want to run their own hypervisors. This is currently supported by nested virtualization, which virtualizes the hardware-virtualize extension (e.g., Intel VT) so that a VM can run its own hypervisor [13]. Nest virtualization not only introduces complexity into the hypervisor design, but also incurs substantial performance overhead. A nested guest in KVM can only reach about 80% of the native performance. For I/O intensive programs, the performance drops to about 25% of the native one. In BM-Hive, users can run their hypervisor of choice (e.g., VMware, KVM, Xen, and Hyper-V) without the additional overhead of nested virtualization.

3 Design

In this section, we first discuss the design requirements for BM-Hive and then present the design of BM-Hive in detail.

3.1 Design Requirements

We have the following design requirements for BM-Hive:

- **Multi-tenancy:** BM-Hive must support multi-tenancy, where a number of bm-guests reside on the same physical server, in order to achieve the fine-grained CPU cores expected by cloud users.
- **Security:** multi-tenancy can lead to serious security concerns. BM-Hive must provide strong hardware-based isolation between bm-guests as well as protect itself from malicious bm-guests. We note that bm-guests are less constrained and thus more powerful than vm-guests.
- **Interoperability:** BM-Hive must be compatible with the existing cloud infrastructure so that to immediately benefit from the availability, reliability, elasticity, and rich features of the cloud infrastructure.
  Interoperability requires that a bm-guest can be run in a VM as well. We call this feature cold migration (as opposite to live migration). Cold migration is a fundamental requirement of the cloud operation. *A prerequisite of cold migration is that bm-guests must be able to connect to the cloud storage and network with high-performance*, because most guests are not allowed to access the local storage. In virtualization, this is supported by para-virtualized I/O such as virtio [30]. BM-Hive solves this challenge with its unique hardware-software hybrid virtio subsystem. From the user perspective, they only need to provide a VM image, which can be run as either a VM or a bm-guest.
- **Performance:** bm-guests should have native CPU and memory performance and close to native I/O performance. Bm-guests should have minimal performance variations.
- **Cost efficiency:** bm-guest should lower cost compare to vm-guest with similar configuration.

Among these requirements, interoperability has the decisive implication on the system design of BM-Hive, as shown in the design overview below.

3.2 System Overview

Fig. 2 compares the architect of virtualization and BM-Hive servers. The main difference is that a vm-guest runs on the virtualized CPU and memory; while a bm-guest runs directly on the physical CPU and memory of its compute board. Both systems utilize virtio for I/O operations. All the major OSes support virtio as an in-kernel or available device driver. The

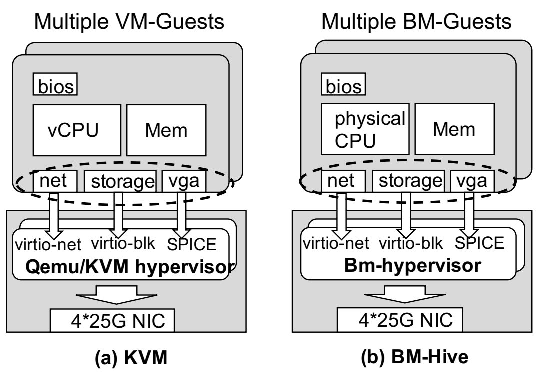

Figure 2. Architecture of BM-Hive, compared to KVM

bm-hypervisor, which is also a user-space process similar to vm-hypervisor, is responsible for managing the life cycle of bm-guests (e.g., assignment, creation, and destruction), providing the backend support for virtio devices, and interfacing with the cloud infrastructure. Because the bm-hypervisor supports the same cloud interface as the vm-hypervisor, it can seamlessly integrate into the existing cloud infrastructure. The code base of the bm-hypervisor is much simpler than that of the vm-hypervisor because the bm-hypervisor does not need to virtualize CPU, memory, buses, etc; Every bm-hypervisor process provides service to one bm-guest only for better isolation of back-end virtio resource. In addition, the bm-hypervisor only interacts with bm-guest indirectly through the virtio interface; while the vm-hypervisor is directly exposed to (malicious) vm-guests through virtual components and hypercalls. The bm-hypervisor is therefore more secure than the vm-hypervisor.

Use scenario: The cloud infrastructure selects an available bare-metal server and picks an idle compute board and powers it on (by turning on the PCIe power). The firmware (i.e., BIOS) on the board then starts executing the boot loader, which will further load the bm-guest kernel. Note that most guests in the cloud are not allowed to use local storage, but have to use the remote cloud storage through the network interface. As such, the bootloader and kernel (both are a part of the VM image) are stored remotely and only accessible through the virtio-blk interface. To address that, we extend the (EFI-based) firmware of the compute board to recognize and utilize virtio during boot. When the kernel is loaded, it can access cloud storage and network through virtio devices.

3.3 System Architecture

Fig. 3 shows the overall system architecture of BM-Hive. Each bare-metal server consists of the base and a number of compute boards. The base is essentially a simplified Xeon-based server with 16 cores E5 CPU. Each compute board is a PCIe extension board to the base. Its main components include the CPU, memory, PCIe bus, and IO-Bond. IO-Bond is a piece of hardware implemented in FPGA. It interfaces with both PCIe buses on the base and the compute board. On the PCIe bus of the compute board, it emulates a number of virtio devices. From the bm-guest's perspective, each virtio device is a normal PCIe device that can be discovered, configured, and used as one. It just happens that these devices are supported by the virtio kernel driver. On the PCIe bus of the base, IO-Bond exposes the backend of the virtio devices and provides the interface for the bm-hypervisor to control the execution of the compute board. IO-Bond acts as a bridge between the virtio front-end in the bm-guest and the back-end in the bm-hypervisor. Specifically, it forwards the control and configure commands by the front-end to the back-end and shuffles the data between them via a built-in DMA engine. Currently, IO-Bond supports the virtio devices for the network and storage (block). It can be easily extended to support other virtio devices with only minor changes (because the main logic of the new device is included in the front- and back-end. IO-Bond only needs to add the PCIe configure space for the new device. The rest can be reused.) IO-Bond can also be implemented by a custom ASIC chip.

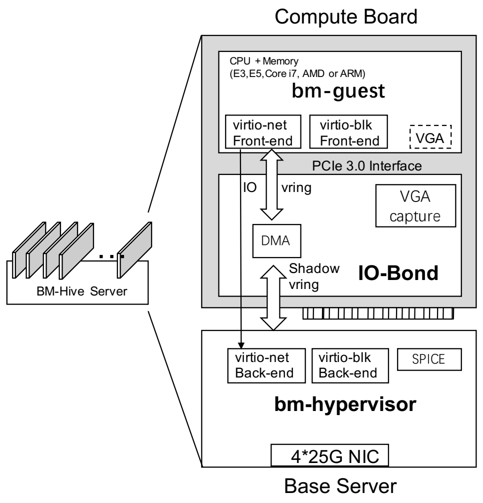

Figure 3. System architecture of BM-Hive

From bm-hypervisor's perspective, each compute board is a special PCIe device for executing bm-guests. A base server in our current implementation can support up to 16 compute boards, i.e., 16 bm-guests can co-reside securely on the same bare-metal server. The main functionality of the bm-hypervisor is threefold: ① it manages the life cycle of the bm-guests. Specifically, it needs to manage the allocation of the compute boards, and control their execution by communicating with IO-Bond. ② The bm-hypervisor provides the back-end support to all the virtio devices of its bm-guests. It maintains the internal states for the virtio devices, and executes the I/O requests from the bm-guests through the network interface. The base CPU has sufficient number of CPU cores to handle all the I/O requests from the bm-guests without losing data. ③ The bm-hypervisor interfaces with the existing cloud infrastructure to integrate itself into the whole cloud operation. Compared to the vm-hypervisor, the bm-hypervisor is not a more-privileged layer sitting beneath the guests. It controls the guests externally. As mentioned before, its code base is much simpler without the need to create a virtual computer system to run the guest. The other components of the bm-hypervisor, including the I/O back-end and cloud interface, are mostly the same as the vm-hypervisor. In fact, they can share the same code base as the vm-hypervisor for easy maintenance.

BM-Hive's architecture not only allows it to be seamlessly integrated into the existing cloud infrastructure, but also makes the design of the compute board highly flexible – the only hard requirement is for it to support the virtio interface. Particularly, it can feature any CPU suitable for its design. We have experimented and produced compute boards with Xeon E3 and E5, Intel Core i7, and Intel Atom processors. Technically, AMD and ARM processors can also be used. Compute boards with Core i7/i9 processors can achieve better single-thread performance than Xeon processors, which is critical for services such as stock trading.

In the rest of this section, we describe the design of IO-Bond in detail.

3.4 IO-Bond

IO-Bond acts as the proxy for the virtio devices of the compute board. Virtio was designed as a general para-virtualized device model usable by different hypervisors. It consists of a front-end and back-end. The front-end sends (high-level) I/O requests to the back-end through a hypervisor-specific notification mechanism (e.g., hypercalls). After the request has been served, the back-end notifies the front-end by injecting a virtual interrupt. The I/O data is exchanged through a shared ring buffer to avoid unnecessary memory copies. Shared buffers are easy to set up on the virtualization server because the front- and back-end can access the same memory. Virtio devices are modeled as PCI devices. A vm-guest can discover, configure, and use virtio devices through its virtual PCI/PCIe bus. This allows BM-Hive to implement virtio in a hardware and software hybrid design.

3.4.1 IO-Bond: Frontend. The front-end of IO-Bond is attached to the PCIe bus of the compute board. The FPGA logic in IO-Bond emulates a PCI interface (i.e., PCI configure space, BAR0, BAR1, PCIe Cap, etc) for each virtio devices. This PCI interface allows the bm-guest to discover, configure, and drive the virtio devices. Access to the PCI interface by the bm-guest is directly forwarded to the back-end for processing.

In the virtualization system, the front- and back-end exchange data through a shared ring buffer. However, this design does not work in BM-Hive because the front- and

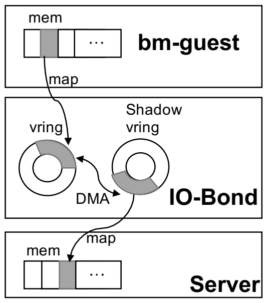

Figure 4. Shadow ring buffers in BM-Hive

back-end of IO-Bond do not share the physical memory. Instead, they each have their own memory. To address that, IO-Bond creates a ring buffer with both the bm-hypervisor and bm-guest. The ring buffer with the bm-hypervisor (shadow vring in Fig. 4) is synchronized to the other ring buffer. When the data is added to one ring buffer, it is copied to the other buffer by the DMA engine in IO-Bond.

3.4.2 IO-Bond: Backend. The back-end of IO-Bond handles the received I/O requests by packaging and forwarding them to the corresponding cloud devices. The design of back-end focus on how to connect our universal cloud infrastructure with high I/O performance. Similar to vm-guest, in BM-Hive, all the I/O requests are handled in the user space with vhost-user protocol interfacing to cloud infrastructure: the customized DPDK vSwitch [1] and the SPDK cloud storage [5]. We uses poll mode driver (PMD) for both DPDK and SPDK. PMD polls the virtio devices for I/O requests instead of relying on interrupts. It can significantly improve the I/O performance by avoiding the interrupt latency, especially when the device runs on the full speed. As such, all the I/O requests are handled completely in the user space. This avoids the extra memory copies (between the user-space and kernel buffers) when sending/receiving packets from the network interfaces. Furthermore, BM-Hive supports a VGA device for users to connect to the console of the bm-guest.

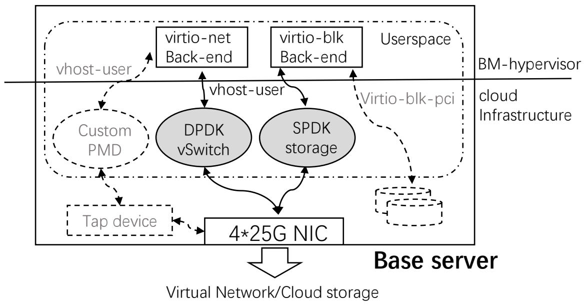

Figure 5. Backend of BM-Hive

We also implemented a few slow I/O paths to bypass cloud infrastructure for testing purposes, e.g., to send packets through the Linux Tap devices. These paths are not deployed in the real cloud due to their low performance or inability to access the cloud services. Only the fast I/O paths with DPDK and SPDK are deployed in our bare-metal servers.

3.4.3 IO-Bond Implementation. IO-Bond is the hardware implementation of virtio PCI interface and counterpart software back-end. The compute board accesses the IO-Bond front-end with standard virtio protocol. Every virtio queue of emulated virtio device has its corresponding back-end shadow vring. The bm-hypervisor communicates with IO-Bond with a pair of mailbox registers for PCI accessing notification and a pair of head/tail registers for each shadow vring. These shadow vrings are actually shared buffers between IO-Bond and bm-hypervisor, they are organized in a list of buf descriptions and map to IO-Bond as shadow vring.

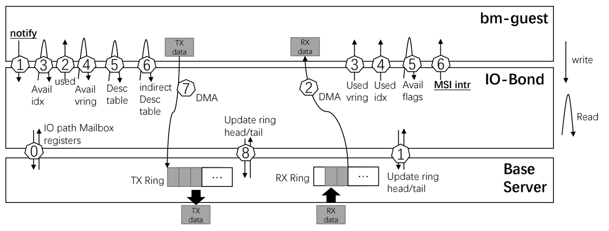

Figure 6. Tx/Rx example of IO-Bond

Fig. 6 shows a typical virtio Tx/Rx workflow how IO-Bond responses guest. Bm-guest notifies IO-Bond by writing to its virtio notification register, and get a MSI interrupt once Rx data arrived. While there are no interrupts between IO-Bond and its back-end. A dedicated thread in bm-hypervisor will pull the updated value of IO mailbox and shadow vring head/tail registers. The example shows 14 steps to complete a Tx send and a Rx read from bm-guest. Step1-6 are those standard virtio device operations including how IO-Bond update vring used-flag, get desc and indirect desc tables. Finally, IO-Bond notifies bm-hypervisor by updating its' head register. Receiving Rx buffer is reverse.

Due to a low cost FPGA used in IO-Bond, a PCI read-/write from bm-guest to IO-Bond front-end takes 0.8 µs, and another 0.8 µs from IO-Bond to its mailbox registers. So a typical PCI access emulating from bm-hypervisor takes 1.6 µs constantly. IO-Bond exposes a PCIe x4 interface each for the virtio network and storage devices. They are backed up by a PCIe x8 interface to the bm-hypervisor. Meanwhile, IO-Bond internal DMA throughput is around 50Gbps. As such, the maximum bandwidth for each bm-guest is 50Gbps (each x4 interface is 32Gbps). As mentioned earlier, all the I/O requests are eventually forwarded to the cloud services through the server's shared (100Gbit/s) network interface. Our cloud infrastructure will balance the needs of all the bm-guests, dynamically adjusts the bandwidth consumption of each bm-guest to ensure fairness.

3.5 Costs efficiency

The profitability in datacenter mainly rely on how many vCPU cores available to be sold with same rack space, in another word, the density of available vCPU cores. A typical vm-based server nowadays chooses two 24cores(48HT) E5 CPUs with 8HT reserved for hypervisor and its host kernel, thus remains only 88HT for users. While with the same rack space, BM-Hive can service up to 8 bm-guests with each 32HT, total 256HT for sell. Even we add additional FPGA (Intel Arria low cost FPGA) for each bm-guest and a base server CPU (which is much cheaper 16HT E5 CPU), BM-Hive still have overwhelming pre-vCPU cost efficiency than vm-guest. Our sell price shows that bm-guest is 10% lower than vm-guest with same configuration.

Power consumption is anther aspect regarding to cost efficiency. However it is very difficult to compare between BM-Hive and vm-based server as they provide different available vCPUs. The most BM-Hive configuration close to vm-based server is a single compute board who sell 96HT (as list in Table 3), while vm-based server sell 88HT instead. Our TDP (Thermal Design Power) estimation shows: BM-Hive with single board has 3.17Watts/per-vCPU, while vm-based server is 3.06Watts/per-vCPU according to Intel processor's TDP[4]. The additional consumption comes from the FPGA hardware and base server' CPU in BM-Hive.

4 Evaluation

In this section, we evaluate the performance of BM-Hive by comparing it to similarly configured virtual and physical machines and with a few applications in the real cloud.

4.1 Environment Setup

BM-Hive has been been deployed and publicly available in our cloud. Table 3 shows the configurations of these instances. Note that the I/O performance of a cloud instance is commonly rate-limited to prevent the misuse of resources and improve overall quality of service. For example, the Xeon E5-2682 instance is limited to 4M packets per second (PPS) and 10Gbit/s in bandwidth for network access and 25K I/O per second (IOPS) for storage access. Most cloud servers feature high-end Xeon processors that have low frequencies; while low-end Xeon processors have higher frequencies but fewer cores [2]. As such, the bare-metal service like BM-Hive is the only option for users that require high single-thread performance.

4.2 CPU and Memory Performance

All the experiments were conducted on the Xeon E5-2682 v4 instance with same Centos-based Linux system. We compared BM-Hive's performance to that of the VM and the physical machine (when applicable). Both the bm-guest and the vm-guest run on the Xeon E5-2682 v4 CPU with 64GB of RAM. VM-guests are exclusive instance and pinned to the physical CPU cores with NUMA node affinity.

Comp Board Type	Core & Freq	Mem	PPS	IOPS	Max
Xeon E3-1240 v6	4core/8HT 3.7GHz	32GB	2M	25K	16
Xeon E5-2682 v4	16core/32HT 2.5GHz	64GB	4M	25K	8
Xeon E5-8163 Platinum	48core/96HT 2.5GHz	384GB	4.5M	25K	1

Table 3. Bare-metal instances available in our cloud. The last column shows the maximum number of the compute boards in a single BM-Hive server. This number depends on the server's power supply, internal space, and I/O performance.

Native CPU and memory performance is critical to users of the bare-metal service. To quantify this performance of BM-Hive, we compared it to that of the vm-guest and the physical machine. The bm-guest and vm-guest had the same CPU and memory configuration (Xeon E5-2682 v4 and 64GB of RAM). However, the physical machine had two sockets of this CPU and 384GB of RAM. [3] They all run the same operating system created from one VM image. The kernel version was 3.10.0-514.26.2.el7. Fig. 7 and 8 show their CPU and memory performance as measured by SPEC CINT 2006 and STREAM benchmarks, respectively. All the benchmarks were run multiple times with consistent results.

The CPU performance of three systems were close to each other. The overall performance of BM-Hive was about 4% faster than the physical machine; while the performance of VM was about 4% slower than the physical machine due to the overhead incurred by virtualization. We note the CPU performance of bm-guest and the physical machine is not exactly the same because they have different configurations and were designed and produced by different manufacturers. The overhead of the vm-guest was attributed to world switches caused by memory virtualization and other events because some SPEC benchmarks are memory intensive. We would like to mention that, even though the CPU performance of the vm-guest is close to the bm-guest, BM-Hive can use CPUs with much higher single-thread performance than common virtualization servers. For example, Xeon E3-1240 v6, the another available instance, is 31% faster in single-core performance than Xeon E5-2682 v4 [8].

The advantage of BM-Hive also shows in the memory performance, as measured by STREAM version 5.1.0. The benchmark was configured to use 1.5GB of memory per array (200M elements, 8Bytes each) and 4.5GB in total. We run the benchmark ten times with 16 threads. The results is shown in Fig. 8. For all the tests, the memory bandwidth of BM-Hive was almost identical to the physical machine, both close to the speed limit of the four memory channels. However, the best performance of the vm-guest can only reach about 98% of the bm-guest under load because of the

[2]Xeon E3 is more closer to desktop processors in the performance and architecture than to Xeon E5 processors.

[3]We could not find a server with the exact same configure. However, the difference in the configuration should not change the overall results.

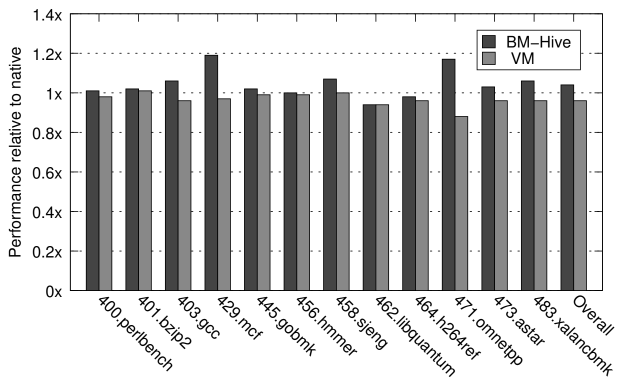

Figure 7. CPU performance by SPEC CPU2006

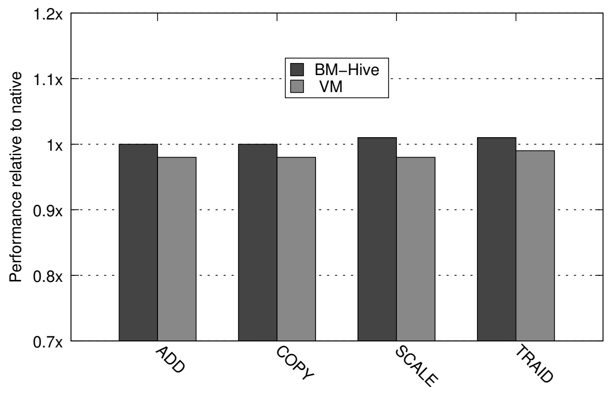

Figure 8. Memory bandwidth by STREAM multi-thread

memory virtualization overhead. The bm-guest does not have this issue because its memory is accessed natively.

Summary: the bm-guest has slightly better CPU performance than the similarly configured vm-guest. However, BM-Hive can utilize CPUs with higher single-thread performance. The memory performance of the bm-guest is substantially better than the vm-guest under load.

4.3 I/O Performance

The bm-guest and vm-guest share most of the same virtio-based I/O path, with the difference that the virtio of BM-Hive has a hardware-software hybrid design. The I/O path of the vm-guest has been aggressively optimized to minimize the overhead. In this section, we compare the network and storage performance of the bm-guest to that of the vm-guest in the cloud environment. Both guests use the virtual cloud network and the cloud storage. As mentioned earlier, the guests are rate-limited to ensure fairness and prevent abuse. Specifically, the network access is limited to 4M PPS and 10Gbit/s in bandwidth, and the storage is limited to 25 IOPS and 300MBps in bandwidth. Therefore, the question we need to answer is whether the bm-guest can fully utilize the allocated network and storage capacity.

Network performance: a key criterion for network performance is how many (small) packets a device can receive / send per second, or the PPS. To measure it, we started two bm-guests on the same server and used netperf-2.5 to send and receive small UDP packets (headers + one byte of data) between them. For comparison, we used two instances of the vm-guest on the server having two Xeon E5-2682 v4 CPUs and 384 GB of memory. As such, the server had sufficient resource to run two vm-guests simultaneously without resource conflicts. We run this test locally to avoid interference on the physical network. The result is shown in Fig. 9. Both the bm-guest and vm-guest reached more than 3.2M PPS. The vm-guest performed slightly better than the bm-guest with less jitters. This was caused by the longer I/O path of BM-Hive than the vm-guest: packets between two vm-guests were exchanged directly through the main memory, but packets between two bm-guests were sent from the source bm-guest to the bm-hypervisor and then forwarded to the destination bm-guest, traversing three PCIe buses.

To understand the IO-Bond overhead, we also compared latency using pair of bm-guests and pair of vm-guests with different tools. Fig. 10 shows the 64bytes UDP latency using sockperf-3.5[11] with default network stack, it was almost same between two type of guests. Meanwhile with DPDK tool[2] to bypass kernel stack, vm-guest was slightly better than BM-Hive due to longer I/O path of BM-Hive. The same thing happens on ICMP ping too.

Another important criterion for the network performance is throughput. In this test, we started two bm-guests on two servers interconnected by a 100Gbit/s network. The vm-guests were similarly configured. The throughput was measured with netperf-2.5 using 64 TCP connections to the server; each TCP packet was 1400Bytes. Both bm-guest and vm-guest reached the full limit of the network bandwidth – 9.6 Gbit/s for the bm-guest and 9.59Gbit/s for the vm-guest.

Overall, the network performance of BM-Hive is more than sufficient for its design purpose. Furthermore, we measured the maximum network performance of BM-Hive by removing the limit on the PPS. Under the same conditions, BM-Hive can achieve 16M PPS, significantly higher than the 4M PPS limit.

Storage performance: In the cloud, storage is normally accessed through the network. To measure its performance, we run the fio benchmark to access the SSD-backed cloud storage through the 100Gbit/s network. We run fio-3.1 with 8 threads and the 4KB data size for random read and write. Both the bm-guest and vm-guest saturated the storage limit, i.e., 25K IOPS. However, the bm-guest had lower average latency and 99.9th percentile latency, as shown in Fig. 11. Specifically, the bm-guest was about 25% faster than the vm-guest in average, and three times faster in the 99.9th percentile latency (for random read). The 99.9th percentile is the latency sample that is larger than 99.9% of all the samples.

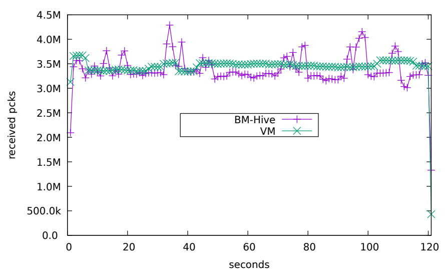

Figure 9. UDP packet receive rate

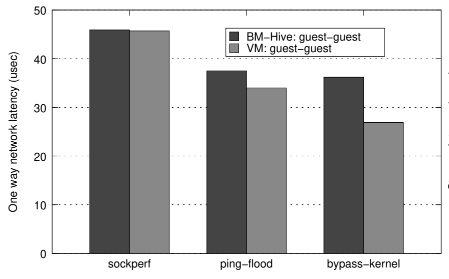

Figure 10. UDP and ping latency

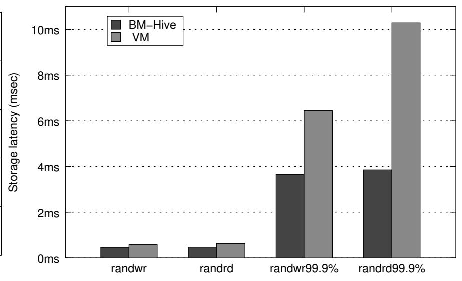

Figure 11. Storage I/O latency

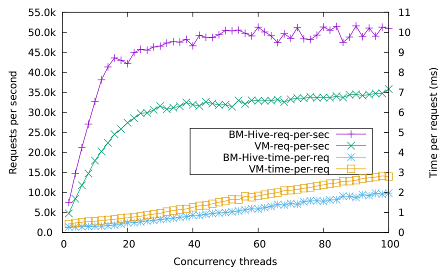

Figure 12. NGINX

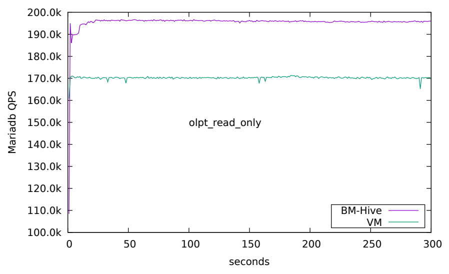

Figure 13. MariaDB ready-only

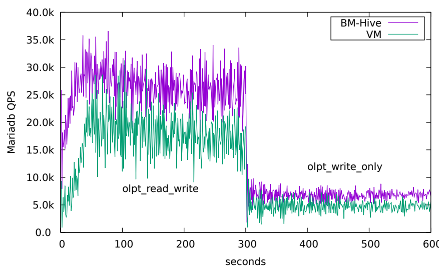

Figure 14. MariaDB rd/wr and wr-only

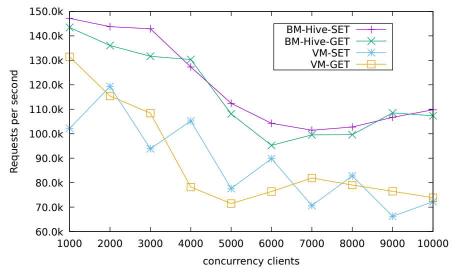

Figure 15. Redis with varying clients

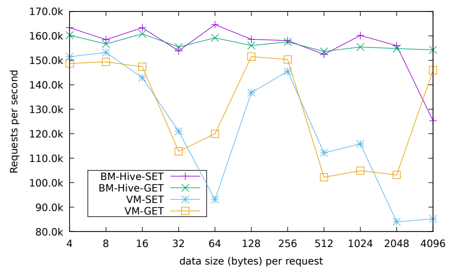

Figure 16. Redis with varying data size

It indicates the worst case performance in latency. It is an important criteria for the I/O performance.

We also measured the unrestricted storage performance of BM-Hive when accessing the local SSD disk. BM-Hive is 50% faster in IOPS and 100% faster in bandwidth than the vm-guest. The average latency is only 60µs.

Overall, the storage performance of BM-Hive is substantially better than the VM because its data are copied directly to the block device's I/O request queue by the DMA engines of IO-Bond; while the vm-guest requires extra memory copies by the CPU.

Summary: both the bm-guest and vm-guest can reach the limits of the cloud-based network and storage I/O. However, the bm-guest provides the better worse-case (99.9 percentile) performance, and performs significantly better than the vm-guest when these limits are lifted.

4.4 Application Performance

An more important experiment is how well BM-Hive performs for common applications used in the cloud. To this end, we run NGINX, MariaDB, and Redis servers in the bm-guest and measured how fast they could respond to queries. Specifically, we used the Apache HTTP benchmark to test the NGINX server with the KeepAlive feature disabled. The result is shown in Fig. 12. When the number of clients increased, bm-guest consistently served about 50% to 60% more requests per second than vm-guest. The average response time per request was about 30% shorter for bm-guest.

The test database for MariaDB contained 16 tables, each with 1 million records. We used sysbench-1.0.17 with 128 threads to query the database simultaneously. The result is shown in Fig. 13 and 14. For read-only queries, the bm-guest sustained 195K queries per second (QPS), while the vm-guest with the same configuration only reached 170K

QPS. i.e., the bm-guest was about 14.7% faster than the vm-guest. In addition, the bm-guest was about 42% faster than the vm-guest in write-only queries and 55% faster in read/write mixed queries.

Redis is an in-memory data structure store. It supports many data structures, such as hashes, lists, and sets. It is often used in the cloud as fast cache for the database. In this experiment, we compare the performance of bm-guest and vm-guest when running the Redis server. We used the standard Redis benchmark as the client and configured the server with 10M random key-value entries. In each test, we queried the server 1M times to get/set the data. Each test was repeated 10 times. The results are shown in Fig. 15 and 16. Specifically, Fig. 15 shows how the servers performed with varying number of clients (from 1,000 to 10,000). The performance of the bm-guest (requests per second) was about 20% to 40% better than that of the vm-guest; Fig. 16 shows how the servers performed with the varying data sizes (from 4B to 4KB). The bm-guest not only processed more requests per second but also had more stable throughput. The fluctuation of the vm-guest performance was likely caused by the cache. Note that the y-axis of Fig. 16 starts with 80K requests-per-second for better visibility.

Summary: BM-Hive performs substantially better than the virtualization-based cloud service for the popular applications used in the cloud.

5 Related Work

In this section, we compare BM-Hive to the closely related work, including efforts to reduce the virtualization overhead and to provide bare-metal and high-density services.

Reducing world switches: the virtualization overhead is mainly caused by world switches, memory and device virtualization, and the hypervisor itself. A world switch is a context switch between the vm-guest and the hypervisor. It is an expensive operation because of the large amount of states that need to be saved and restored. World switches can be caused by many events, such as memory virtualization and timers. A number of approaches have been proposed to reduce the number and frequency of world switches. For example, most modern hypervisor intercepts all the hardware interrupts (even for devices passed through the guest). High-speed devices like network interface cards can generate a lot of interrupts. To this end, ELI (Exit-Less Interrupt) proposes to remove the hypervisor from the interrupt handling path and let the guest directly and securely handle interrupts [18]. By doing so, ELI can significantly improve the throughput and reduce the latency for I/O-intensive workloads. Interrupts on bm-guests are handled natively and thus have no additional overhead. Related to reducing world switches, there are efforts to improve the responsiveness of the guest. For example, KVM recently introduced the halt_polling feature, which poll for wake conditions before yielding the CPU and entering the sleep state. This avoids the problem where the guest is put to sleep and immediately waken up by the pending wake conditions [20]. Another example is to use co-scheduling [6, 32] to solve the lock-holder preemption problem [33], where the guest is preempted while holding a lock. All these problems do not exist in BM-Hive.

Memory virtualization: memory is usually virtualized by two-level page tables – the guest page table (GPT) maps the guest virtual address to guest physical address, and the extended (or nested) page table (EPT) maps the guest physical address to the physical address. A TLB (Translation Lookaside Buffer) missing in the guest is thus substantially more expensive to handle than that in the bm-guest because it need to traverse two levels of page tables(up to 24 memory accesses [31]). Solutions have been proposed to increase the TLB coverage (e.g., by using huge pages) or to reduce TLB penalty. For example, POM-TLB proposes to reduce the cost of address translation by using a very large TLB that is a part of the memory [31]. Chang et. al proposes a hardware-software hybrid design to coalesce contiguous pages into a subset of page table entries, thus increasing the coverage of TLB entries [29]. Amit proposes to reduce unnecessary TLB shootdown, a process to maintain the coherency of (per-core) TLB caches, by tracking the page access [12]. These issues do not affect BM-Hive because its memory is accessed natively. It can reach the native performance for memory.

Device virtualization: device nowadays is mostly virtualized by para-virtualization in which a custom device driver is loaded into the guest kernel to directly handle I/O operations. Virtio is the most popular para-virtualized I/O interface. It is natively supported by Linux. In addition, some hardware devices natively support virtualization, such as SR-IOV network adapters. Such devices can be directly passed-through to the guest. BM-Hive uniquely implements the virtio interface with a hybrid approach so that it is interoperable with the existing cloud infrastructure.

Minimal hypervisor: there are also efforts to create minimal hypervisors, often by partitioning the system resources among VMs. For example, Intel's ACRN is a minimal hypervisor for embedded IoT systems [25]. It partitions the CPU cores and the memory directly to VMs to reduce CPU and memory overhead. NoHype proposes to eliminate the hypervisor by partitioning the resources among VMs by the hardware [23]. Bm-hypervisor of BM-Hive is much simpler than the traditional hypervisors because it does not virtualize anything and has no direct interaction with the bm-guests.

Nest hypervisor: one of the goals of BM-Hive is to allow users to run their own hypervisors, an increasingly important use case of cloud computing. This can theoretically be supported by nested virtualization [13, 19], where the guest hypervisor runs inside a virtual machine. However, nested virtualization has high performance overhead [13]. BM-Hive provides a more efficient solution to this problem since the user's hypervisor runs directly on the physical CPU and has full control over the hardware virtualization support.

Other bare-metal services: Users have long been able to lease physical machines in datacenters. However, machine leasing is not considered a bare-metal service because it is not integrated into the cloud infrastructure, and thus lacks the benefits of the cloud service. For example, it could take hours, if not days, to setup the lease machine because the user has to provide a physical hard disk to be installed on the machine. In BM-Hive, bm-guests can use the same VM images for both bare-metal and virtualized services. The scheduling of the service is instant.

Most public cloud vendors have deployed single-tenant bare-metal services, with the limitations mentioned in Section 1. Compared to single-tenant bare-metal services, BM-Hive has higher density, is more cost-efficient and can use CPUs with high single-thread performance (which is often unavailable in the cloud).

Some public cloud vendors also developed specific network device for VM. For example, Azure has deployed SmartNIC [17] on their datacenter to offload most of software defined network functions into hardware. From our best knowledge, SmartNIC focus on optimization of low level cloud infrastructure, while BM-Hive do not touch cloud infrastructure layer.

There are some systems with similar overall architecture. For example, VCA2, Intel's media codec processor, is a PCIe card with three Xeon E3 v1258L processors [9]. Each server can install up to eight VCA2 cards to speed up the media encoding/decoding. BM-Hive is built for the cloud service. Its hardware design is geared toward this purpose. For example, it features a hybrid virtio interface to allow bm-guests to interact with various cloud services.

6 Discussion

In this section, we discuss potential improvements to BM-Hive and the future research plans.

Improvements to IO-Bond: our current design of IO-Bond is rather straightforward. We can put more intelligence into it to improve the overall system design. For example, we plan to add more network-related functions in IO-Bond to offload the packet processing from the bm-hypervisor so that lower-cost CPUs can be used by the base.

IO-Bond is currently implemented by FPGA. After the design and function of IO-Bond stabilize, it can be implemented by ASIC chips to save cost and improve performance. We estimate a 75% reduction in the PCI response time from 0.8µs to 0.2µs for guest requests by IO-Bond.

Live Migration and Upgrade: VM live migration is an important tool for the cloud provider. It can migrate a VM from one physical server to another while the VM is running. With VM live migration, the cloud provider can upgrade anything on the original physical server, from hardware devices to the whole software stack. However, VM live migration does not scale well in the whole cloud data center because it relies on backup servers to temporarily hold VMs during the live migration. To address that, a recent system, Orthus, proposes to live-upgrade the VMM (e.g., KVM + QEMU) [34] without halting the VMs.

The design of BM-Hive makes it straightforward to apply the live upgrade approach proposed in Orthus because it is mostly a subset of the full VMM software stack. However, it is more challenging to live-migrate bm-guests. Technically, we can insert a virtualization layer into the bm-guest at run-time and convert the bare-metal guest to a special vm-guest, which can then be migrated to another compute board [21, 28]. We have built a working prototype of this design. However, there are two drawbacks with this design. First, the cloud provider is not supposed to access or change cloud users' systems. This approach is thus too intrusive. Second, the injected virtualization layer has to make assumptions about the user system, such as the OS it is running, making the approach difficult to work for all bm-guests.

SGX Support: SGX is a trusted execution environment from Intel that only needs to trust the CPU itself to guarantee secrecy and security. SGX is becoming increasingly popular for cloud users from finance, stock trading, and e-commerce sections. The current design of SGX does not work well in virtual machines. For example, the KVM hypervisor and QEMU require special builds with the SGX SDK and the guest kernel requires additional drivers [7]. We plan to add native support to SGX in BM-Hive so that users can directly migrate their SGX code to the bare-metal service without additional efforts.

7 Summary

We have presented the design, implementation, and evaluation of BM-Hive, a high-density bare-metal cloud service. In BM-Hive, each bm-guest runs on a PCIe card with its own CPU and memory. It can seamlessly interact with the existing cloud services through the hybrid virtio interface. Our evaluation demonstrates the performance advantage of BM-Hive over the traditional VM.

8 Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments that helped improve this paper.

References

[1] 2017. Data Plane Development Kit. https://www.dpdk.org/. Online; accessed April 17th, 2019.

[2] 2017. Data Plane Development Kit: basicfwd example. https://github.com/DPDK/dpdk/blob/master/examples/skeleton/basicfwd.c. Online; accessed April 17th, 2019.

[3] 2017. Intel Chip Flaws Leave Millions Of Devices Exposed. https://www.wired.com/story/intel-management-engine-vulnerabilities-pcs-servers-iot/. Online; accessed April 17th, 2019.

[4] 2017. Intel® Xeon® Platinum 8160T Processor. https://ark.intel.com/content/www/us/en/ark/products/123543/intel-xeon-platinum-8160t-processor-33m-cache-2-10-ghz.html. Online; accessed Dec 25nd, 2019.

[5] 2017. Storage Performance Development Kit. https://spdk.io/. Online; accessed April 17th, 2019.

[6] 2018. Coscheduling for Linux. http://lkml.iu.edu/hypermail/linux/kernel/1809.0/06774.html?print=cGhvcm9uaXgK. Online; accessed April 17th, 2019.

[7] 2018. SGX Virtualization on QEMU/KVM. https://github.com/intel/qemu-sgx. Online; accessed April 17th, 2019.

[8] 2019. CPU Benchmarks, Single-thread Performance. https://www.cpubenchmark.net/singleThread.html. Online; accessed April 17th, 2019.

[9] 2019. Intel Visual Compute Accelerator 2. https://www.intel.com/content/www/us/en/products/servers/accelerators.html. Online; accessed April 17th, 2019.

[10] 2019. Linux Kernel CVEs: All Indexed CVEs. https://www.linuxkernelcves.com/cves. Online; accessed April 17th, 2019.

[11] 2019. sockperf. https://github.com/mellanox/sockperf/. Online; accessed April 17th, 2019.

[12] Nadav Amit. 2017. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *Proceedings of the 2017 USENIX Conference on Annual Technical Conference (USENIX ATC'17)*.

[13] Muli Ben-Yehuda, Michael D Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. 2010. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation.*

[14] Muli Ben-yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert Van Doorn. 2007. The Price of Safety: Evaluating IOMMU Performance. In *In Ottawa Linux Symposium (OLS.*

[15] 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 Conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 273–286.

[16] Yaakoub El-Khamra, Hyunjoo Kim, Shantenu Jha, and Manish Parashar. 2010. Exploring the Performance Fluctuations of HPC Workloads on Clouds. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CLOUDCOM '10)*. IEEE Computer Society, Washington, DC, USA, 383–387. https://doi.org/10.1109/CloudCom.2010.84

[17] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: SmartNICs in the public cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 51–66.

[18] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. 2012. ELI: Bare-metal Performance for I/O Virtualization. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 411–422. https://doi.org/10.1145/2150976.2151020

[19] Alexander Graf and Joerg Roedel. 2009. Nesting the Virtualized World. Linux Plumbers Conference.

[20] halt polling [n. d.]. The KVM halt polling system. https://www.kernel.org/doc/Documentation/virtual/kvm/halt-polling.txt.

[21] Jaeseong Im, Jongyul Kim, Jonguk Kim, Seongwook Jin, and Seungryoul Maeng. 2017. On-demand Virtualization for Live Migration in Bare Metal Cloud. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. ACM, New York, NY, USA, 378–389. https://doi.org/10.1145/3127479.3129254

[22] Intel. 2018. Intel Side Channel Vulnerability L1TF. https://www.intel.com/content/www/us/en/architecture-and-technology/l1tf.html.

[23] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. 2010. NoHype: Virtualized Cloud Infrastructure Without the Virtualization. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 350–361. https://doi.org/10.1145/1815961.1816010

[24] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. *Spectre Attacks: Exploiting Speculative Execution*. Technical Report. https://arxiv.org/abs/1801.01203

[25] Hao Li, Xuefei Xu, Jinkui Ren, and Yaozu Dong. 2019. ACRN: A Big Little Hypervisor for IoT Development. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2019)*. ACM, New York, NY, USA, 31–44. https://doi.org/10.1145/3313808.3313816

[26] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[27] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks Are Practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 605–622. https://doi.org/10.1109/SP.2015.43

[28] Yushi Omote, Takahiro Shinagawa, and Kazuhiko Kato. 2015. Improving Agility and Elasticity in Bare-metal Clouds. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 145–159. https://doi.org/10.1145/2694344.2694349

[29] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. 2017. Hybrid TLB Coalescing: Improving TLB Translation Coverage Under Diverse Fragmented Memory Allocations. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 444–456. https://doi.org/10.1145/3079856.3080217

[30] Rusty Russell. 2008. Virtio: Towards a De-facto Standard for Virtual I/O Devices. *ACM SIGOPS Operating Systems Review* 42, 5 (2008).

[31] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. 2017. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 469–480. https://doi.org/10.1145/3079856.3080210

[32] Jan H. Schönherr, Bianca Lutz, and Jan Richling. 2012. Non-intrusive Coscheduling for General Purpose Operating Systems. In *Proceedings of the 2012 International Conference on Multicore Software Engineering, Performance, and Tools (MSEPT'12)*. Springer-Verlag, Berlin, Heidelberg, 66–77. https://doi.org/10.1007/978-3-642-31202-1_7

[33] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. 2004. Towards Scalable Multiprocessor Virtual Machines. In *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3 (VM'04)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=1267242.1267246

[34] Xiantao Zhang, Xiao Zheng, Zhi Wang, Qi Li, Junkang Fu, Yang Zhang, and Yibin Shen. 2019. Fast and Scalable VMM Live Upgrade in Large Cloud Infrastructure. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 93–105. https://doi.org/10.1145/3297858.3304034

[35] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 990–1003. https://doi.org/10.1145/2660267.2660356