VBMq:Pursuit Bare-metal Performance by Embracing Block I/O Parallelism in Virtualization

Paper ID: 455

***Abstract*—Barely acceptable block I/O (Input/Output) performance prevents virtualization from being widely used in the HPC (High-Performance Computing) field. Although the Virtio paravirtualized framework brings great I/O performance improvement for virtualization, there is a sharp performance degradation when accessing high-performance NAND-flash devices in virtualized environment due to their data parallel design. The primary cause of this fact is deficiency of block I/O concurrency in virtualized environment. In this paper, we propose a novel design of block I/O mechanism called VBMq (Virtio Block Multiqueues) for virtualization, which aims to solve the block I/O concurrency issue in virtualization. VBMq is a paravirtualized driver for block devices based on Virtio framework. It bypasses QEMU global mutex and utilizes multiple individual I/O threads to handling I/O requests simultaneously. In the meanwhile, it assigns a non-overlapping CPU to each I/O thread in order to eliminate mutex contentions among I/O threads. In addition, we also set CPU affinity to optimize each I/O completion path for every request. The CPU affinity setting is very helpful to reduce CPU cache miss rate and increase CPU efficiency. We implement the prototype system based on Linux 4.1 kernel and QEMU (Quick Emulator) 2.3.1. Our experiments show that VBMq scales graciously with multi-core environment, and the block I/O performance in a guest machine is close to that of a bare-metal.**

I. Introduction

Virtualization for High-Performance Computing (HPC) has been intensively exploited in recent years, because of its scalability, reliability, fault tolerance [][1], [2]. However, barely acceptable block Input/Output (I/O) performance prevents virtualization from being widely used in the HPC field. Because the storage requirements of HPC are significantly different from traditional server and workstation workloads. High IOPS, low latency, concurrent processing and consistent reliability have become the indispensable requirements of secondary storage devices in the HPC world [3].

In order to satisfy the concurrent processing requirement of HPC in data storage, Bjorling *et al*.[4] proposed a novel design of Multi-Queue Block I/O Queueing Mechanism (*blk-mq*) for Linux block layer. The *blk-mq* eliminates mutex contention in kernel and utilize the concurrency mechanism in many-core environment by using two level queues: software staging queues and hardware dispatch queues. The software staging queues can be configured such that there is one such queue per core on the system. Therefore, each software queue can take advantage of a dedicated core on the system to handle block I/O requests and completions concurrently. In the meanwhile, the underlying secondary storage devices with parallel data design can achieve their optimal performance.

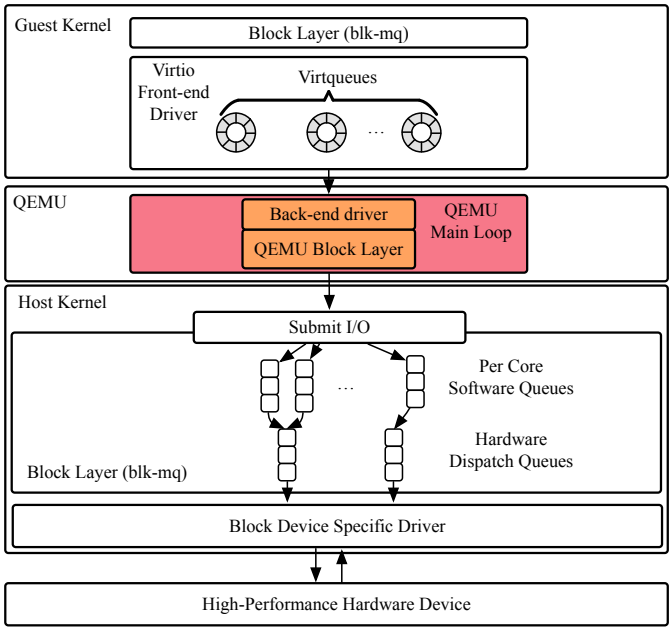


Fig. 1. Architecture of KVM with virtio-blk paravirtualization driver and there is no concurrent processing ability in QEMU block layer. All I/O events in the VM have to pass through QEMU block layer constrained by the main loop (red region) in turn.

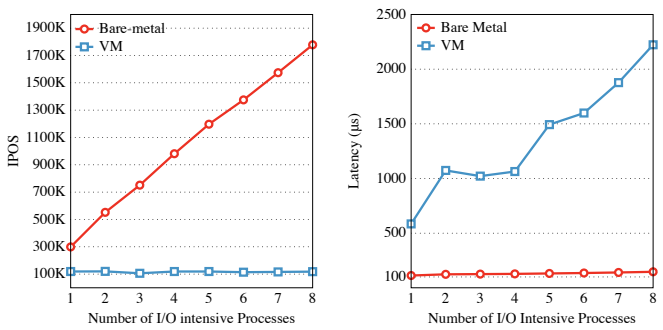
Unfortunately, the *blk-mq* is limited in the virtualized environment due to lack of support for concurrency in hypervisor [5]. Figure 1 illustrates the block I/O workflow from a VM (Virtual Machine) to the underlying storage device in Linux KVM (Kernel-based Virtual Machine) [6] with *virtio-blk* [7] paravirtualized driver. KVM is a virtualization infrastructure that turns Linux kernel into a hypervisor, and it uses Quick Emulator (QEMU) [8] to perform hardware emulation. The overall block I/O workflow path in Figure 1 can be divided into three parts from top to bottom: 1. guest block I/O layer, 2. QEMU block I/O layer and and 3. host block I/O layer. The guest block I/O layer is the same as the host block I/O layer because they are both Linux kernel block layer. Therefore, except for the middle part, the other two parts are able to handle I/O requests and completions concurrently due to *blk-mq* is adopted by them. However, with the purpose of thread-safe, all jobs are done in a single main loop (the default event loop thread in QEMU) when QEMU submits block I/O requests to the host on behalf of the guest. The main loop and vitrual CPU (vCPU) threads share the QEMU global mutex to execute QEMU code in turn. Which means no matter how

many concurrent I/O threads in the guest block I/O layer, only one I/O thread in the QEMU block layer is allowed to submit I/O requests to the host block layer at any given time. Lack of support for handling block I/O requests concurrently makes QEMU become a bottleneck in the block I/O workflow path for virtualization. Thus, there is much more work need to be done to achieve concurrent processing for block I/O in virtualized environment.

In this paper, we propose an advanced block I/O approach to address the concurrency issue in virtualization. Our approach is based on *virtio* paravirtualization framework for block I/O in KVM which is a Linux hypervisor, called *virtio* block multi-queues (VBMq). It relies on multiple I/O threads to not only bypass the global mutex in QEMU but also handle I/O requests concurrently without delay. It is widely noticed that the current QEMU in Linux suffers from severe mutex contention when submitting I/O requests, which becomes the primary bottleneck to the block I/O in virtualized environment. This mutex design is necessary for thread-saft, but especially painful on many-core environment. It incurs big mutex contention problem, which apparently leads to significant performance degradation, because all cores must accord with the state of the lock. Our approach solves these disadvantages of KVM paravirtualized block I/O by redesigning the QEMU block layer in three ways. The first is to adopt a new kind of block I/O thread which can avoid the mutex contention inside QEMU. The second is to build multi-thread mechanism for submitting I/O requests between I/O submission queues in the paravirtualized device driver and software staging queues in the host kernel, which enables the VM to communicate with the host block layer directly for optimal performance. The third is to simplify I/O path for reducing workflow overhead.

The proposed prototype of VBMq is based on *virtio* paravirtualized framework, which consists of the front-end driver in the guest OS and the back-end driver in QEMU. As a software approach, it means that VBMq is applicable to any architecture *e.g.* PPC64, X86. We implemented the prototype based on Linux 4.1 kernel and QEMU 2.3.1. And we conducted detailed experiments on both a null block device and a physical high-performance storage device in our test platform. The null block device can simulate a virtual storage device by receiving block I/O requests and acknowledging completions instantly. And we further demonstrate the proposed system by experiments on real devices. Notice that, Although our implementation is on QEMU and KVM, it is not limited the specific system but rather generally applicable. We only take advantage of the fact that KVM uses an asymmetric model in which some of the code is virtualized while other codes (*e.g.* , the native storage system) run on bare-metal unaware of the existence of the hypervisor. The benchmark results show that the performance of the VM can catch up with that of the bare-metal when accessing the high-performance NAND-flash storage device.

The contribution of this paper is threefold: First, We performed a benchmark-driven analysis of virtualization overheads in block I/O. Second, A novel design of VBMq was proposed based on the analysis. And finally, the evaluation



(a) IOPS (b) Latency

Fig. 2. Benchmark analysis on the VM with *virtio-blk* block device driver and the bare-metal system (FIO, 4KB, random read direct I/O, 32 iodepth and libaio), presents the large performance gap between the VM and the bare-metal.

proves that the prototype system of VBMq achieves stable performance improvement for high-performance NAND-flash storage device which is widely used in HPC filed.

The rest of the paper is organized as follows: Section II explains the background performance gap in the current native KVM paravirtualization block I/O layer. Section III presents the architecture of VBMq. Section IV describes the experimental methodology, and then detailed benchmark results and performance evaluation are presented in Section V. Related work is introduced in Section VI. Finally, we conclude our approach in Section VII.

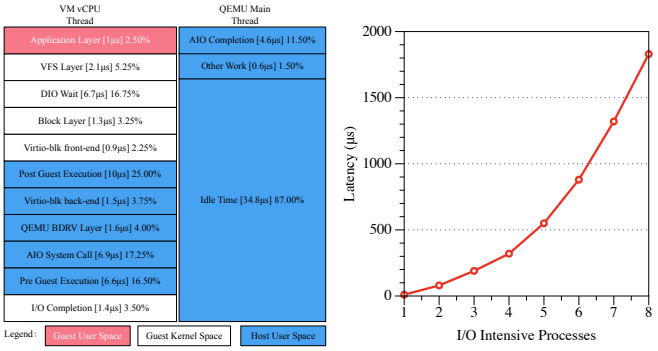
II. Motivation

To the end of better understanding the proposed design, in this section, firstly, we introduce background about block I/O virtualization. Secondly, we introduce the detail of the benchmark-driven analysis of virtualization overheads in block I/O for KVM. And finally, we present the challenges of unblocking the bottleneck in block I/O for virtualization.

A. Block I/O Virtualization Primer

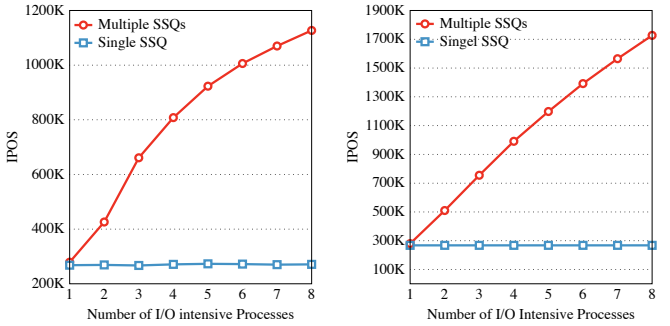
Hypervisor is a middleware that, on one hand, allows VMs to access diverse storage devices in a uniform way. And on the other hand, provides storage devices and drivers with a single point of entry from VMs. There are mainly two software approaches for accessing block devices in virtualized environment. One is full emulation where the hypervisor fully emulates a specific block device in software [9]. The operation system resides in the VM (guest OS) and accesses the emulated block device by its regular device driver. Although the full emulation does not require any change in the guest OS, the performance is very poor.

On the contrary, the other method, paravirtualization, significantly improves the performance. It allows the guest OS to run some specialized codes to cooperate with the hypervisor. And thus improves the I/O performance. For example, *virtio* is a paravirtualized framework for KVM, which presents several ring buffers transport organized as one or more *virtqueues* and device configuration as a PCI (Peripheral Component



(a) Single HDQ (b) Multiple HDQs

Fig. 3. 4KB direct I/O block I/O path tracing, alone with associated latency. The latency is dramatically increased due to lack of concurrent processing support in QEMU.



(a) Single HDQ (b) Multiple HDQs

Fig. 4. Benchmark analysis on the *blk-mq* in the bare-metal (FIO, 4KB, random read direct I/O, 32 iodepth and libaio), SSQ denotes software staging queue and HDQ denotes hardware dispatch queue.

Interconnect) device. The paravirtualized block driver, *virtio-blk*, is implemented by *virtio*. It places pointers to buffers on the *virtqueue* and uses Programmed I/O (PIO) command to initiates block I/O requests. Moreover, the efficient is further improved because the hypervisor can directly access *virtqueues* from the memory of guest OS without copying (zero-copy [10]). Paravirtualized will require driver installaton on guest OS, nevertheless, paravirtualized devices perform much better than fully emulated devices. This is mainly because data can be exchanged between the VM and the hypervisor through batch process. Becaused of it, paravirtualization is one of the most widely used approach for I/O virtualization.

B. Benchmark-driven analysis of paravirtualization

High-performance NAND-flash storage deivces are widely used in HPC filed to satisfy the requirement of HPC applications. To evaluate the effectiveness of paravirtualization on KVM when accessing the high-performance NAND-flash device, we employed the null block device to simulate the high-performance NAND-flash device. We performed comparative tests on the VM which has 20 vCPU (virtual CPU) and bare-metal system respectively. The tests used FIO (Flexible I/O Tester) to generate block I/O workloads for measuring.

From the benchmark tests, we observed a large performance gap between the VM and the bare-metal system, as shown in Figure 2(a). The more I/O intensive processes we executed, the bigger the performance gap we got. Figure 2(b) depicts the latency of the block I/O from the VM are proportionate to the number of I/O intensive processes. While the latencies in the bare-metal system seems to be irrelevant to the number of I/O intensive processes.

For further investigation, we traced the major code blocks when executing a 4KB direct I/O read request submitted by the VM using *virtio-blk*. The purpose is to pin point the bottleneck. According to the tracing result, we found out frequent global switches (*exit* and *entry*) between the VM and the hypervisor. As shown in Figure 3(a), the guest OS kernel first handles the I/O requests initiated by the guest application. Then, the requests pass through VFS layer and arrive at the Direct I/O (DIO) waiting time which consists of global switches and context switches between threads running in the guest OS. When I/O requests pass through block layer in the guest kernel space, the *virtio-blk* front-end driver iterates over them in the elevator queue and places I/O description of each request on the *virtqueue*. Then a PIO command initiated by the front-end driver leads to a global switch from the VM to the hypervisor, control is forwarded to the KVM module to handle the *exit*. Afterwards KVM identifies the cause of the *exit* and passes control to the virtio-blk back-end. It extracts the I/O descriptors from the virtqueue without copies and passes the requests to the QEMU block driver layer (QEMU BDRV). After initiating Asynchronous I/Os (AIO) to the block device, the VM resumes execution. Finally, an event-driven dedicated QEMU thread receives I/O completions and forwards them to the *virtio-blk* back-end driver. The back-end driver updates the *virtqueues* with completion information and calls upon KVM to inject an interrupt into the guest OS, for which KVM must initiate an *entry*. After resumption, the guest kernel handles the interrupt as normal, and then signals the end of interrupt, causing another switch.

Notice that latency of *post guest execution* was the longest (25% of the total). It accounts for the work done by both QEMU and KVM to switch contexts between the guest OS and QEMU. Especially, compared to the other code blocks depicted in Figure 3(a), there was a striking increase of delay for *post guest execution* when we executed more block I/O intensive processes concurrently, as shown in Figure 3(b). Because the work done by *post guest execution* is postponed by the severe mutex contention for I/O events in QEMU.

In conclusion, the benchmark results reveal two overheads in I/O performance for virtualization. The first one is the lack of concurrency for I/O thread, which is due to the architecture of hypervisor. Because there is only one I/O thread for each VM allocated in KVM. Its capacity is insufficient to fully take advantage of optimal performance for the storage device with data parallel design. The second one is high latency in virtualized environment. The latency in the VM are dozens of times than those of the bare-metal system because, on the one

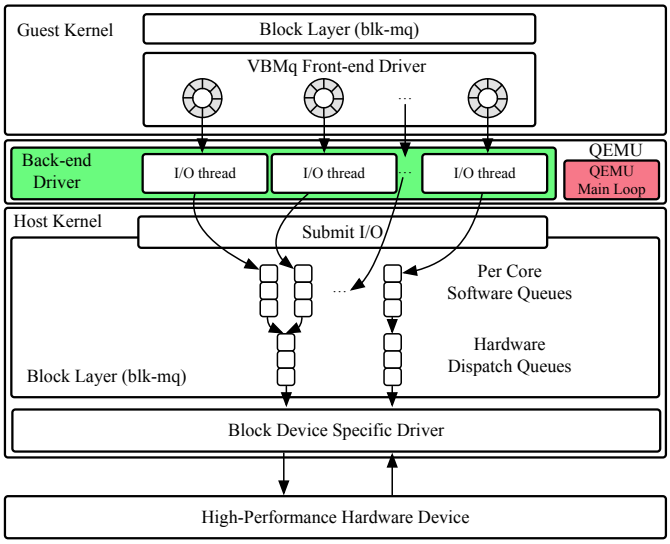


Fig. 5. Design of VBMq, all I/O threads in back-end driver (the green region) are independent with QEMU event loop (the red region). The I/O threads in the green region bypass the QEMU main loop, which perform high IOPS compared to the native QEMU block layer.

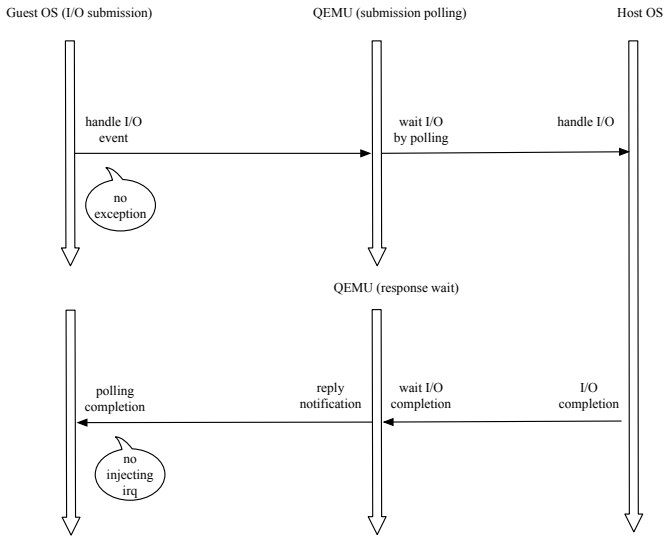


Fig. 6. Polling based I/O threads. The polling threads is able to handle I/O requests without triggering global switches between the VM and the hypervisor.

hand, the I/O workflow of the VM is almost two times than that of the bare-metal, on the other hand, the mutex contention triggered by concurrency issue of I/O thread exacerbates the delay.

C. Challenge of Optimization

There are two challenges for us to unblock the bottleneck in block I/O for virtualization. The first is how to eliminate global mutex. Because the lack of support for multiprocessing is a fundamental reason which lead to performance degradation in virtualized environment. To QEMU-specific, all vCPUs and the main loop have to use the QEMU global mutex to serialize execution of core code of QEMU for thread-safe. Therefore, making I/O threads independent with global mutex in virtualization is the most important thing needed to be concerned.

The second is how to establish optimal concurrency mechanism for I/O threads. In the bare-metal block layer, two level queues are responsible for keeping I/O concurrency. while the software staging queues have direct impact on performance compared to the hardware dispatch queues. Figure 4 depicts the performance comparison on two level queues. No matter how many hardware dispatch queues the kernel block layer has, the number of software staging queues determines the upper limit for I/O performance. Therefore, if we want to achieve optimal performance for the storage device in virtualization, the interaction between concurrent I/O threads and software staging queues in the host block layer should be established.

III. Architectue and Implementation

Based on the results of benchmark analysis, we propose the design of VBMq to unblock bottleneck for block I/O in virtualization. Figure 5 illustrates the architecture of VBMq, along with block I/O workflow in it. In the following part of this section, we detail our design in four parts including the front-end driver, concurrency mechanism in back-end, configuration of I/O threads and optimization for I/O completions.

A. Design of VBMq Front-end Driver

The VBMq front-end driver is the paravirtualized device driver reside in the guest OS. In order to achieve optimal performance for block I/O in the guest OS, the VBMq front-end driver need to be responsible for hardware dispatch queue registration for block layer in the guest OS. The front-end driver describes the structure and the number of submission queues that it supports for matching hardware dispatch queues in the block layer. The number of submission queues can be set from 1 to 2048 as supported by Message Signaled Interrupts Extended (MSI-X) [11]. Under multiple submission queues condition, we map each submission queue onto a nonoverlapping vCPU directly in order to avoid unnecessary CPU scheduling.

Furthermore, the front-end driver also establishes a mapping between a given software staging queue associated with a vCPU and the befitting hardware dispatch queue.

B. Design of Concurrency

The back-end driver is the most important link of VBMq. It should be able to create multiple I/O threads which provide concurrent transferability for I/O requests from submission queues in the front-end driver to software staging queues in the host kernel without the involvement of QEMU main loop and global mutex. Therefore, there are two types of I/O thread techniques can be used for this purpose.

One is event-driven I/O thread. We know that QEMU is an event-driven program which is capable of doing several things at once using an event loop. When using event-driven

I/O thread technique to create I/O threads, the back-end driver spreads the work of accessing submission queues in the front-end driver across several I/O threads instead of the single main loop. In this case, each I/O thread must run an *AioContext* event loop which originates from the QEMU block layer. While *AioContext* is not thread-safe, so some changes must be introduced if we want to enable multiple I/O threads. Firstly, we open multiple Block Driver States (BDSes) for the same file descriptor. Each BDS is linked to an *AioContext*, which allows the back-end driver to handle I/O submissions in the right *AioContext*. Secondly, we bind each submission queue in the front-end driver to a different BDS. Thirdly, we put each *AioContext* into a separate I/O thread.

The other is polling [12], [13] based I/O threads technique. Most recent studies [14], [15], [16] claimed that overheads in I/O virtualization were mainly caused by context switches between the host and the guest. Therefore, most studies tended to reduce the switch times based on the polling mechanism. By adding polling threads to QEMU, as shown in Figure 6, the submission polling thread polls the address space shard with the front-end driver for checking I/O submission coming from the guest OS. The I/O submissions are transferred to the software staging queues in the host kernel if they are in the front-end driver submissions queues. In the meanwhile, a response wait thread starts to wait for I/O completion in the blocking state. After the submitted I/O is done, the response wait thread wakes up and notifies the guest OS of the I/O completion. Also, the guest OS utilizes the dedicated thread to check the I/O completion by polling, and then completes I/O. In this case, the overall procedure does not notify QEMU, so there is no global switch occurred. In other words, the polling technique is friendly with multi-thread since all polling threads are able to process independently without the involvement of QEMU main loop and global mutex. Furthermore, the latency of each I/O request can be dramatically decreased by configuring CPU affinity to all polling threads.

No matter what kind of I/O thread technique is adopted, each submission queue in the front-end driver can be accessed separately and concurrently by having their own threads. We also configure CPU affinity to each thread for reducing redundant CPU scheduling. Furthermore, each thread directly submits I/O requests to a different software staging queue reside in the host kernel for optimal concurrency.

C. Configuration of I/O Threads

Under event-driven I/O thread condition, the number of I/O threads is the same with the number of submission queues in the front-end driver. Because hardware dispatch queues in block layer of the guest OS should match submission queues in the front-end driver due to the design of our front-end driver. Moreover, the number of software queues is as much as the number of vCPUs assigned to the VM. As we mentioned before, the optimal performance achieved when the number of software queues is at least as much as the number of hardware dispatch queues. Consequently, each submission queue should have a dedicated I/O thread for handling requests in it.

TABLE I
NVM-EXPRESS SSD PARAMETERS

Type	SM1715 NVM-Express SSD
Maximum queue depth	65536 queue; 65536 commands per queue
Uncacheable register access	Two per command
MSI-X and Interrupt steering	2048 MSI-X interrupts
Parallelism and multiple threads	No locking
Efficiency for 4KB commands	Gets command parameters in one 64-byte fetch

Under polling I/O thread condition, each submission queue in the front-end driver has a group of threads for handling requests and completions, such as submission polling thread, completion polling thread. Therefore, the number of I/O threads is three times as much as the submission queues in the front-end driver.

D. Optimization for Multiple IO Completions

Different from polling based I/O threads, multiple context switches in the I/O completion path always causes software delays under event-driven I/O threads condition. Therefore, we also need to optimize the I/O completion process after redesigning I/O submission workflow. Each VBMq device is configured as a PCI device plugged in guest. MSI-X (Message Signaled Interrupts Extended) interrupts for PCI are used by QEMU to notify I/O completions when I/O requests are executed by the host kernel. Although MSI-X has ability to deal with the interrupts effectively on multi-core environment, it is actually useless in QEMU due to the single I/O thread that is responsible for submissions and completions. In this situation, if a large number of I/O completions wait to be processed on the same CPU, the average latency of I/O completions will become longer. Moreover, that submissions and completions are handled on different CPUs must lead to a amount of cache miss. In view of all this, we need to configure CPU affinity for I/O completions in VBMq. Consequently, to a specific I/O thread, a dedicate CPU which is identical with the one assigned to it during the submission period is also assigned to it for handling the I/O completion. It is very helpful to reduce cache miss rate and increase CPU efficiency without redundant scheduling.

IV. EXPERIMENTAL METHODOLOGY

We implemented a prototype of VBMq (around 2000 lines code), as described in the previous section, within the KVM hypervisor. In this section, we mainly detail our experimental methodology for validating the performance of VBMq.

A. Platform

The test platform is an IBM Power System S822L (8247-22L), equipped with is a dual-socket, 10-cores-per-socket IBM

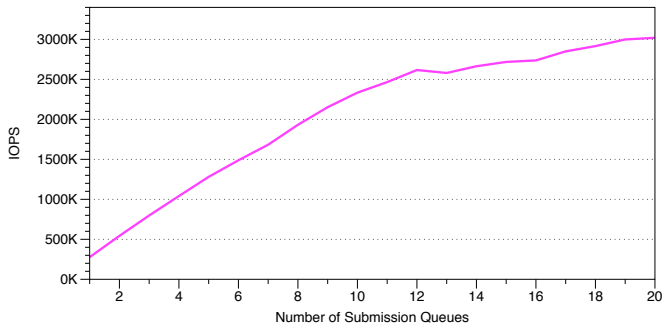


Fig. 7. Evaluate impact of the number of submission queues in the front-end driver using the null block device (FIO, 4KB, random read direct I/O, 32 iodepth and libaio). With the increase of submission queues, the VM's throughput gradually goes up.

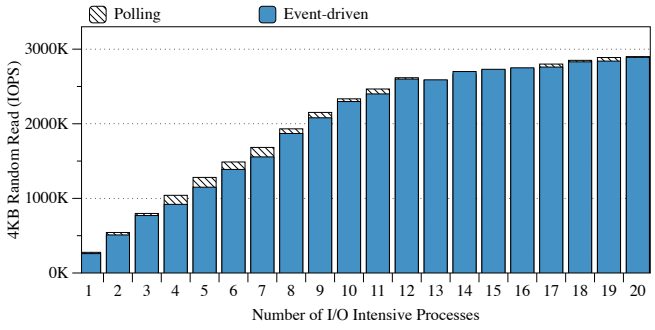


Fig. 8. Performance comparison on two kinds of I/O thread mechanism (FIO, 4KB, random read direct I/O, 32 iodepth and libaio), overall performance of polling is better than that of event-driven.

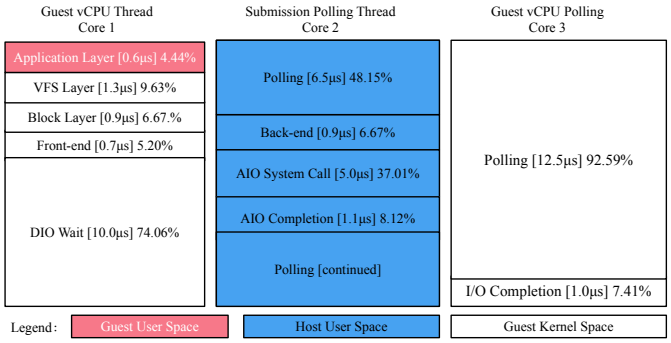


Fig. 9. VBMq block I/O path with polling threads. The result shows that the latency of per I/O request is dramatically decreased by using polling.

POWER8 CPU [17] running at 3.42GHz with Simultaneous Multi-Threading (SMT) disabled. The system includes 192GB of memory, the null block device and a Samsung SM1715 1.6TB NVM-Express soild state drive (SSD) [18] with the Nallatech 385 CAPI card. The VM running on the platform has 20 vCPUs (a single vCPU per core) and 64GB RAM for performance-minded evaluation.

Notice that there are two types of storage devices are used in the evaluation. The null block device use memory to simulate a virtual storage device with parallelism design, which can validate a higher range of performance where the real storage device cannot reach. In the meanwhile, we utilize a Samsung SM1715 1.6TB NVM-Express SSD to enlarge our evaluation scope. The NVM-Express is a logical device interface specification for accessing non-volatile storage media via PCI-Express (PCIe) bus. It allows levels of parallelism found in modern SSDs to be fully utilized by the host hardware and software. Table I indicates technical features of the SM1715. It is able to transfer data concurrently on a multi-core system without synchronizations among each core because it has 2048 MSI-X interrupts and 65,536 queues that each of them has 65,536 commands. Furthermore, the paired submission and completion queue mechanism is used for efficient and improve performance with minimal latency.

The basic software collection used in the evaluation comprises KVM hypervisor based on kernel 4.1 for IBM Power System, QEMU 2.3.1 and a set of benchmark tools like Flexible I/O Tester (FIO) and *ftrace*.

B. I/O Workload Generation and Trace Collection

We focus our evaluations on throughput and latency. We measure throughput by overlapping the submission of asynchronous I/O requests. The I/O workloads for experiment are generated by FIO which is able to carefully control several kinds of configurations like block device, block size, queue-depth, I/O engine, cache mode and the number of I/O processes. Moreover, there are many detailed information such as the number of context switch, throughput, IOPS and latency. These outputs are very helpful for us to analyze the results. In all experiments, each workload runs for 60 seconds in order to collect sufficient data.

We utilize a default Linux kernel trace tool (*ftrace*) to trace the I/O activities at the block layer. The trace data are stored in the *tmpfs* which is a temporary filesystem that resides in memory in order to minimize the interference caused by tracing.

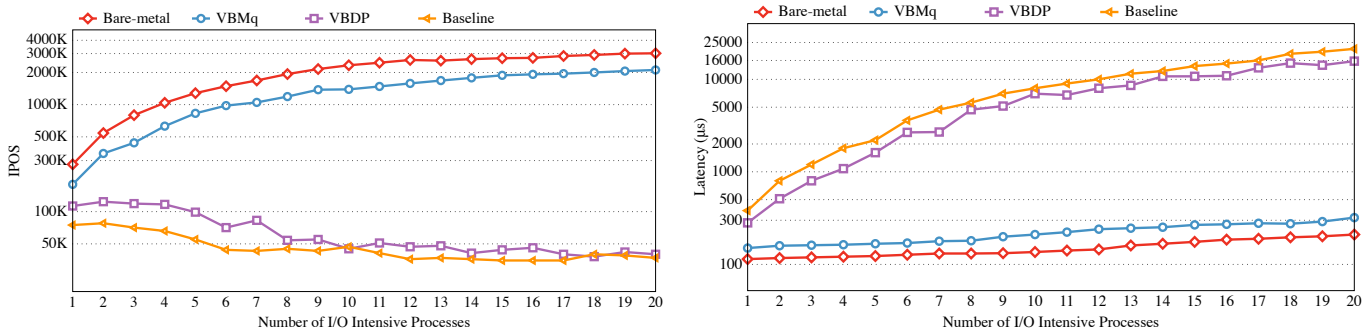
C. Performance Metrics

The primary metrics of our experiments are absolute throughput (IOPS) and latency (μ-seconds) of the block layer. We contrast the results achieved by virtualized and bare-metal environments to prove that the former can approach the latter.

V. Evaluation

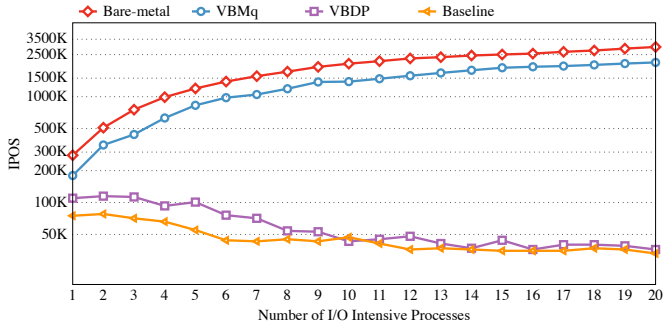
The evaluation has two phases. In the first phase evaluation, we performed experiments on the prototype of VBMq. The influence for block I/O performance by varying the number of submission queues and the effect of the polling I/O thread and the event-driven I/O thread are the points what we are concerned in this phase.

In the second phase evaluation, we compared our new block layer design for virtualization with the native QEMU block layer and virtio-blk-data-plane technique. The virtio-blk-data-plane technique is a previous optimization for the native QEMU block layer which improves the block I/O performance

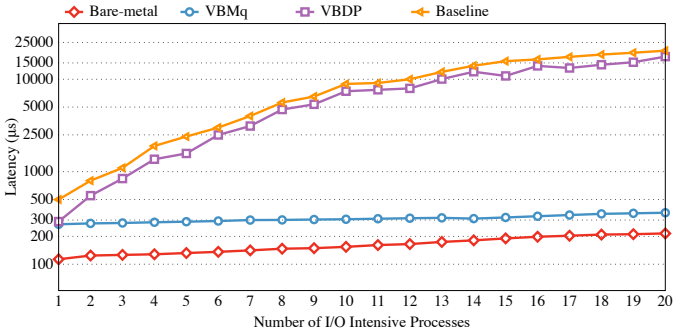


(a) 4KB Random Read Direct I/O

(b) 4KB Random Read Direct I/O



(c) 4KB Random Write Direct I/O



(d) 4KB Random Write Direct I/O

Fig. 10. Performance comparison on two types of I/O workloads among Baseline, VBDP, VBMq and bare-metal with the null block device.

by using a single dedicated I/O thread which avoids the QEMU global mutex for each storage devices. In the remaining of the paper, We denote the native QEMU block layer as **Baseline** and virtio-blk-data-plane technique as **VBDP**.

A. *Impact of the Number of I/O Submission Queues*

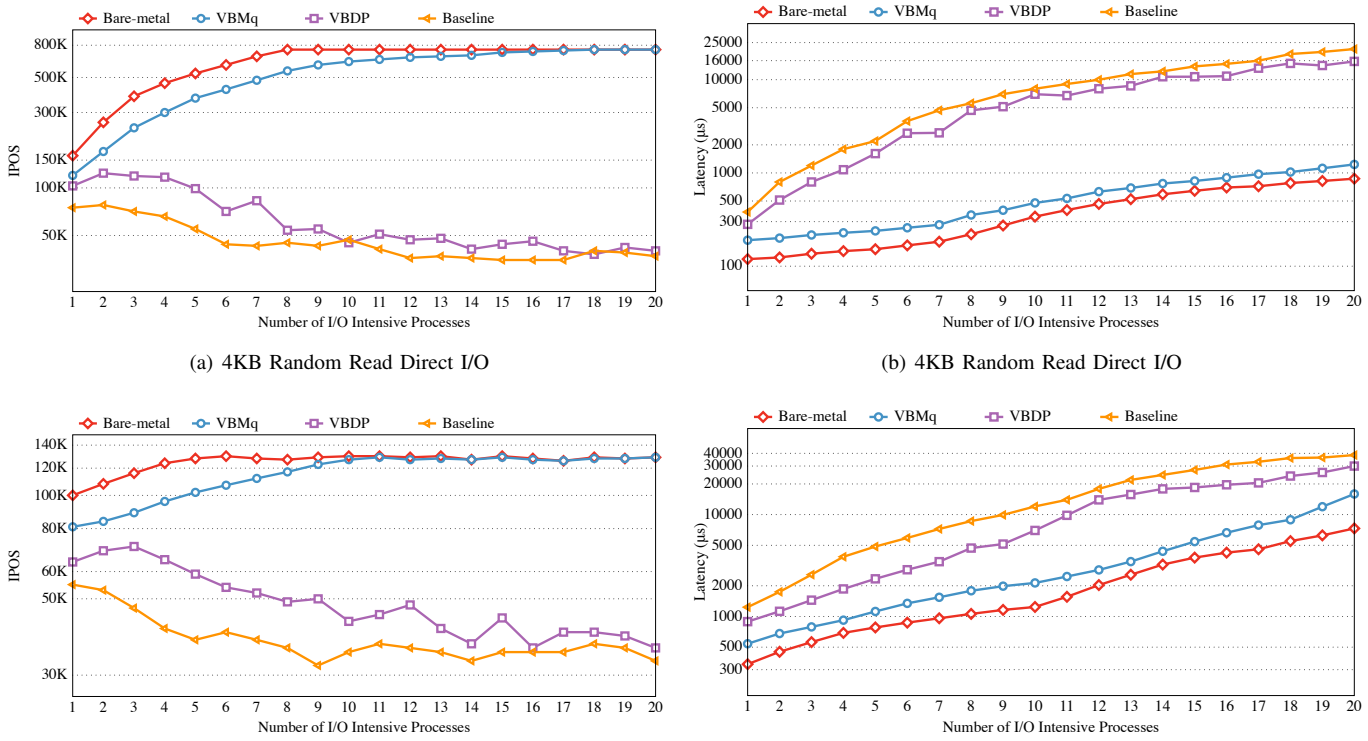
As we mentioned earlier, the number of submission queues in the front-end driver can be configured from 1 to 2046. In order to validate the optimal configuration for the front-end driver, we observed the change of the absolute throughput (IOPS) by varying the number of submission queues. Figure 7 shows that there is consistent growth in IOPS by increasing submission queues. In this comparison, the highest IOPS is achieved when setting the number of submission queues to 20. There is no visible performance improvement even though we feed more submission queues to the front-end driver.

According to our analysis, two mainly reasons result in this phenomenon. One reason is the architecture of *blk-mq*. We know that, with blk-mq, the number of software staging queues is typically as much as the number of cores the system assigned for optimal parallelism. As a result, the throughput will reach the peak if the number of hardware dispatch queues is equal to the number of software staging queues. Notice that just 20 CPUs assigned to the VM in our experiment. Therefore, the optimum configuration of the number of the submission queues is 20. The other reason is the number of I/O threads in the back-end driver. In our design, the I/O threads in the back-end have a consistent one-to-one mapping to the submission queues in the front-end. Thus, more I/O threads enlarge the path from QEMU to the block layer in the host kernel.

B. *Effect of Multiple I/O Threads*

Two types of I/O thread mechanisms are introduced to our design: event-driven I/O thread and polling based I/O thread. We implemented both of them in order to find out which one can achieve better performance. Theoretically, the event-driven mechanism generates more context switches than the polling design, especially when improving the number of I/O submission queues. However, one of the most significant worries about the use of the polling mechanism is more CPU resource consumption than the event-driven mechanism. Because many polling threads may lead to performance degradation when the system is lack of CPU resources.

Figure 8 illustrates the result of the contrast test between these two I/O thread mechanisms. It is clearly that the performance of the polling is better than that of event-driven which decreased by 8%. This is primary due to two reasons. The first reason is because the polling mechanism is capable of removing the interrupt overhead and reduce the context switch cost. By tracing the block I/O path, as shown in Figure 9, The thread polls the request queue for I/O requests coming from the guest machine and for I/O completions coming from the host machine. The polling thread invokes the VBMq back-end on incoming requests and completions. The polling thread is outside QEMU. When it detects I/O completions event, it invokes I/O completion stack by the interrupt handler.



(a) 4KB Random Read Direct I/O

(b) 4KB Random Read Direct I/O

(c) 4KB Random Write Direct I/O

(d) 4KB Random Write Direct I/O

Fig. 11. Performance comparison on two types of I/O workloads among Baseline, VBDP, VBMq and bare-metal with the NVM-Express SSD

We eliminate I/O-related context switch by using poll I/O threads on the submission queues. We do not need to lock the submission queues, because each of them can be accessed by only one polling thread which is different from the others. For a 4KB direct I/O read, the polling mechanism reduce the latency from 40μs to 13.5μs. The second reason is many-core environment. The test platform has 20 independent physical cores, it is convenient for us to set CPU affinity to each polling thread. Therefore, all I/O threads can handle I/O requests and completions as soon as possible without unnecessary CPU scheduling.

Depending on the analysis in the first phase evaluation, the prototype system used in the second phase evaluation adopted the polling based I/O thread mechanism as well as 20 submission queues in the front-end driver.

C. Evaluation on Null Block Device

We measured and compared the performance of the null block device under four different conditions including Baseline, VBDP, VBMq and bare-metal. The I/O intensive processes were generated by FIO using the 2 types of I/O workloads depicted in Section IV.B and we varied the number of I/O intensive processes from 1 to 20 in the experiment. Figure 10(a) illustrates the random read case. Notice that the null block device performs amazing throughput arriving 3 million IOPS, but it is not the ultimate limit of the null block device. Bare-metal's throughput was the optimal performance could be achieved by our test platform. Baseline's throughput gradually decreased with the number of I/O intensive processes increased. Although the VBDP's throughput was a bit higher than Baseline's, it is also on the decline along with the increase of the number of I/O intensive processes. Especially, VBDP's through was almost equal to Baseline's when the number of I/O intensive processes is greater than 18. Only VBMq's throughput goes up as quickly as bare-metal's. When executing 20 I/O intentive processes concurrently, VBMq's throughput had 40-fold increase compared to Baseline and VBDP and there is no more than 32% performance degradation compared to bare-metal. Figure 10(b) depicts the results of block I/O latency in the experiment. VBMq's latency was shorter than VBDP's and Baseline's while a little longer than bare-metal's. The results of the random write case are similar as the random read case, as shown in Figure 10(c), 10(d).

The benchmark results in Figure 10 explain that the VM is able to process block I/O requests concurrently as same as the bare-metal system by using VBMq. However, the performance gap between the VM and the bare-metal is not bridged entirely. It is not caused by the design flaw in our approach, but only due to the longer block I/O path in virtualized environment compared to bare-metal. This overhead is too difficult to eliminate except for redesigning the entire block layer in Linux.

D. Evaluation on NVM-Express Device

In the end of evaluation, we performed the same experiment with a real physical storage device in order to enlarge the

evaluation scope.

Firstly, we tested the SM1715 NVM-Express SSD mounted on our test platform in bare-metal for achieving its maximum throughput. Different from the null block device, the SM1715 has relatively poor through which is limited to 750K random read IOPS and 130K random write IOPS. For this reason, the throughput improvement is stopped when the number of concurrent I/O intensive processes reaching 8. There is no visible increase of throughput when the number of I/O intensive processes increased from 8 to 20, as shown in Figure 11(a), 11(c). In virtualized environment, Baseline's and VBDP's throughput also decreased with the number of I/O intensive processes increased. But the VBMq's throughput achieved the performance cap of the SM1715 when the number of concurrent I/O intensive processes arrived 18.

In conclusion, the VM is capable of fully taking advantage of the underlying high-performance storage device by using VBMq.

VI. Relatedwork

As the hypervisor playground, Linux offers a variety kinds of hypervisor solutions with different advantages and attributes. However, the narrow I/O path, mutex contention and frequent context switch (*exit* and *entry*) have dramatically decreased the I/O performance in virtualized environment [16], [19]. Consequently, most recent studies have concentrated on eliminating overheads of virtualization in block I/O. The effective approaches can be divided into two main categories: hardware acceleration and software optimization.

Numerous manufacturers provide a number of interfaces and technologies for supporting I/O virtualization at the hardware layer like SR-IOV (Single Root I/O Virtualization) [20], IOMMU (I/O Memory Management Unit) [21], [22], [23], Intel VT-d [24], [25] and Steata Storage System [26]. These outstanding methods present superior quality when adopting them into virtualization infrastructure, even the performance is close to that of bare-metal environment, but they could be inappropriate in some condition since they are lack of compatibility to all architectures and expensive specialized hardware is a need. Moreover, some of these hardware ways such as SR-IOV and IOMMU require a dedicated VM (Virtual Machine) to comply with an underlying physical device that leads limitation to the benefits of virtualization like VMs Live migration [27], [28]. Therefore, many subsequent efforts still remain to be done to overcome the limitations of the hardware approaches.

In the meanwhile, Several researchers have surveyed the issues of I/O overheads triggered during communications between the VM and the hypervisor. They have devised some software-based ways such as *virtio* paravirtualized framework, Efficient and Scalable Para-virtual I/O System (ELVIS) [29] and ELI (Exit-Less Interrupts) [30] in order to decrease the I/O overheads in virtualization. It is well known that these remarkable approaches have obtained significant performance improvement by optimizing the block I/O path and relieving mutex contention in virtualized environment.

VII. Conclusion

In this paper, we have established that the current design of the block layer for virtualization is unable to fully utilize the high-performance parallel NAND-flash storage device which is widely used in HPC for high throughput. We proposed a novel design for block layer in virtualization. This design is based on multiple submission queues and multiple I/O threads in order to bypass the global mutex in hypervisor and handle I/O requests concurrently.

We implemented a prototype of our design within the KVM and evaluated the prototype on the null block device, as well as on a physical storage device. The evaluation results have shown the superiority of our design and its scalability on many-core systems for HPC.

References

[1] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis, "Virtualization for high-performance computing," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 2, pp. 8–11, 2006.

[2] W. Huang, J. Liu, B. Abali, and D. K. Panda, "A case for high performance computing with virtual machines," in *Proceedings of the 20th annual international conference on Supercomputing*. ACM, 2006, pp. 125–134.

[3] C. Fang, "Using nvme gen3 pcie ssd cards in high-density servers for high-performance big data transfer over multiple network channels," SLAC National Accelerator Laboratory (SLAC), Tech. Rep., 2015.

[4] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block io: introducing multi-queue ssd access on multi-core systems," in *Proceedings of the 6th International Systems and Storage Conference*. ACM, 2013, p. 22.

[5] J. Liu, W. Huang, B. Abali, and D. K. Panda, "High performance vmm-bypass i/o in virtual machines." in *USENIX Annual Technical Conference, General Track*, 2006, pp. 29–42.

[6] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, 2007, pp. 225–230.

[7] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[8] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[9] B. H. Lim *et al.*, "Virtualizing i/o devices on vmware workstation\'s hosted virtual machine monitor," 2001.

[10] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa, "Pin-down cache: A virtual memory management technique for zero-copy communication," in *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*. IEEE, 1998, pp. 308–314.

[11] A. Martinez, J. Chapple, P. Sethi, and J. Bennett, "Circuitry to selectively produce msi signals," Jun. 29 2004, uS Patent App. 10/881,076.

[12] O. Maquelin, G. R. Gao, H. H. Hum, K. B. Theobald, and X.-M. Tian, "Polling watchdog: Combining polling and interrupts for efficient message handling," in *ACM SIGARCH Computer Architecture News*, vol. 24, no. 2. ACM, 1996, pp. 179–188.

[13] C. Dovrolis, B. Thayer, and P. Ramanathan, "Hip: hybrid interrupt-polling for the network interface," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 4, pp. 50–60, 2001.

[14] A. Gordon, N. Har'El, A. Landau, M. Ben-Yehuda, and A. Traeger, "Towards exitless and efficient paravirtual i/o," in *Proceedings of the 5th Annual International Systems and Storage Conference*. ACM, 2012, p. 10.

[15] A. Landau, M. Ben-Yehuda, and A. Gordon, "Splitx: Split guest/hypervisor execution on multi-core." in *WIOV*, 2011.

[16] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," *ACM Sigplan Notices*, vol. 41, no. 11, pp. 2–13, 2006.

[17] B. Sinharoy, J. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. Brown, J. E. Moreira *et al.*, "Ibm power8 processor core microarchitecture," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 2–1, 2015.

[18] J. J. Hung, K. Bu, Z. L. Sun, J. T. Diao, and J. B. Liu, "Pci express-based nvme solid state disk," in *Applied Mechanics and Materials*, vol. 464. Trans Tech Publ, 2014, pp. 365–368.

[19] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour, "The turtles project: Design and implementation of nested virtualization." in *OSDI*, vol. 10, 2010, pp. 423–436.

[20] A. Unknown, "Single root i/o virtualization and sharing specification, revision 1.0," *Sep*, vol. 11, pp. 1–84, 2007.

[21] M. Ben-Yehuda, J. Mason, J. Xenidis, O. Krieger, L. Van Doorn, J. Nakajima, A. Mallick, and E. Wahlig, "Utilizing iommus for virtualization in linux and xen," in *OLS06: The 2006 Ottawa Linux Symposium*. Citeseer, 2006, pp. 71–86.

[22] I. AMD, "O virtualization technology (iommu) specification," *AMD Pub*, vol. 34434, 2007.

[23] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. Van Doorn, "The price of safety: Evaluating iommu performance," in *The Ottawa Linux Symposium*, 2007, pp. 9–20.

[24] R. Hiremane, "Intel virtualization technology for directed i/o (intel vt-d)," *Technology@ Intel Magazine*, vol. 4, no. 10, 2007.

[25] J. Che, Q. He, Q. Gao, and D. Huang, "Performance measuring and comparing of virtual machine monitors," in *Embedded and Ubiquitous Computing, 2008. EUC'08. IEEE/IFIP International Conference on*, vol. 2. IEEE, 2008, pp. 381–386.

[26] B. Cully, J. Wires, D. Meyer, K. Jamieson, K. Fraser, T. Deegan, D. Stodden, G. Lefebvre, D. Ferstay, and A. Warfield, "Strata: High-performance scalable storage on virtualized non-volatile memory," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014, pp. 17–31.

[27] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 273–286.

[28] U. Lublin and Q. A. Liguori, "Kvm live migration," in *KVM Forum*, 2007.

[29] N. Har'El, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky, "Efficient and scalable paravirtual i/o system." in *USENIX Annual Technical Conference*, 2013, pp. 231–242.

[30] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir, "Eli: bare-metal performance for i/o virtualization," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 411–422, 2012.