# Understanding the Impact of Cache Locations on Storage Performance and Energy Consumption of Virtualization Systems

*Tao Lu*[§], *Ping Huang*[§], *Xubin He*[§], *Ming Zhang*[‡]
[§] *Virginia Commonwealth University, USA,* [‡] *EMC Corporation, USA*
*{lut2, phuang, xhe2}@vcu.edu, ming.zhang@emc.com*

## Abstract

As per-server CPU cores and DRAM capacity continuously increase, application density of virtualization platforms hikes. High application density imposes tremendous pressure on storage systems. Layers of caches are deployed to improve storage performance. Owing to its manageability and transparency advantages, hypervisor-side caching is widely employed. However, hypervisor-side caches locate at the lower layer of VM disk filesystems. Thus, the critical path of cache access involves the virtual I/O sub-path, which is expensive in operation latency and CPU cycles. The virtual I/O sub-path caps the throughput (IOPS) of the hypervisor-side cache and incurs additional energy consumption. It's viable to directly allocate spare cache resources such as DRAM of a host machine to a VM for building a VM-side cache so as to obviate the I/O virtualization overheads. In this work, we quantitatively compare the performance and energy efficiency of VM-side and hypervisor-side caches based on DRAM, SATA SSD, and PCIe SSD devices. Insights of this work can direct designs of high-performance and energy-efficient virtualization systems in the future.

## 1 Introduction

Virtualization techniques are the backbone of production clouds. As CPU cores and memory capacities increase, virtual machine density of production clouds increases, disk storage becomes a salient performance bottleneck. DRAM or SSD-based caches [1–4] have been deployed in virtualization storage stacks for I/O acceleration of guest virtual machines. Currently, caches are commonly implemented in hypervisors due to several reasons. First, hypervisor-side caches are transparent to VMs, thus, can be naturally ported across many guest operating systems. Second, a hypervisor has complete control over system resources, therefore, can make the most informed cache management decisions such as cache capacity allocation and page replacement [5]. Third, comparing with VM-side caches, hypervisor-side caches can be shared, resulting in increased resource utilization [2].

Despite the management flexibilities, hypervisor-side cache access involves the costly I/O virtualization layers, which cause intensive activities of the userspace process such as *qemu*, and incur considerable interrupt deliveries [6]. Study on *virtio* [7] with network transactions shows that a busy virtualized web-server may consume 40% more energy, due to 5x more CPU cycles to deliver a packet, than its non-virtualized counterparts [8]. Similarly, our tests on storage transactions show that I/O virtualization caps hypervisor-side cache performance and increases per I/O energy consumption. In contrast, VM-side caches may obviate the I/O virtualization overheads. More specifically, for 4KB read requests at the IOPS of 5k, hypervisor-side DRAM caches consume about 3x the power and delivers 30x the per I/O latency of VM-side DRAM caches. For a server hosting four I/O intensive VMs, the hypervisor-side caching consumes 48 watts while the VM-side caching consumes only 13 watts. The idle power of the server is 121 watts. In other words, comparing with hypervisor-side caching, VM-side caching can save about 25% of the entire server's active power. Unfortunately, benefits of VM-side caching have long been ignored. To rouse awareness of its benefits, as a first step, we conduct empirical comparisons between hypervisor-side (host-side) and VM-side (guest-side) caches. We believe answering the following questions is helpful for future cache designs:

- How much are the performance and energy efficiency penalties for hypervisor-side caching?
- What are the root causes of the penalties of hypervisor-side caching?
- Why do these penalties need to be mitigated?
- What are the potential approaches to reduce these penalties?

In this paper, we present empirical studies of these problems on a *KVM* [9] virtualization platform using DRAM, SATA SSD, and PCIe SSD as cache devices. Our evaluation demonstrates that VM-side DRAM

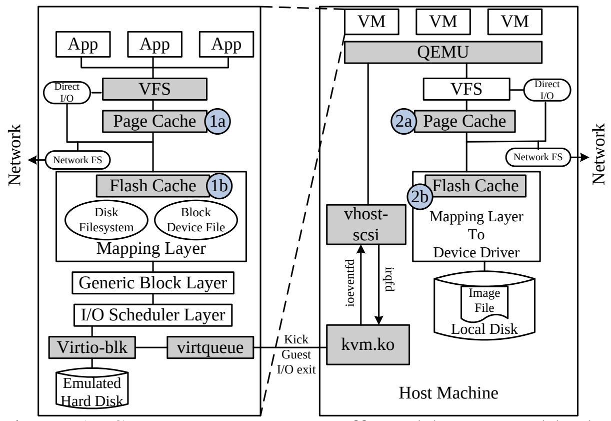


Figure 1: System components affected by a VM block device operation. DRAM-based page caches are built-in modules of operating systems. Flash-based caches are optional but widely deployed in virtualization platforms to accelerate VM storage.

caches yield enticing performance and energy efficiency gains over their hypervisor-side counterparts. For a SATA SSD, either attached to a VM to build a VM-side cache, or used to build a hypervisor-side cache, *virtio* is involved in the device access path. As a result, VM-side SATA SSD caches don't provide benefits over their hypervisor-side counterparts. However, *PCI passthrough* enables a VM bypassing the virtual I/O path and directly accessing a PCIe SSD in a dedicated way to achieve better performance and energy efficiency.

## 2 Problem Analysis

In this section, we first present storage access for VMs, explain the involvement of storage components when cache hits at VM and hypervisor side. Then, we analyze where the virtual I/O overheads lie in.

### 2.1 Storage Access for VMs

Block devices are commonly exposed to VMs via emulation. The host-side entity of a virtualized block device can be a file, an LVM logical volume, a device partition, or a whole device. Since device emulation incurs overheads, dedicated device allocation, also known as device passthrough, is implemented to enable a device being exclusively used by a VM without the involvement of I/O virtualization layers. However, not all devices can be allocated in the way of passthrough. For block devices, currently only the PCI-based devices such as PCIe SSDs can be assigned to VMs via *PCI passthrough*; SATA devices, however, cannot be allocated to VMs in the way of passthrough. In this paper we assume the persistent storage of VMs is backed by emulated block devices. But when we discuss the implementations of SSD-based guest-side caches, we will compare the performance and energy efficiency of SSDs connected to VMs via *PCI passthrough* and *virtio*, respectively.

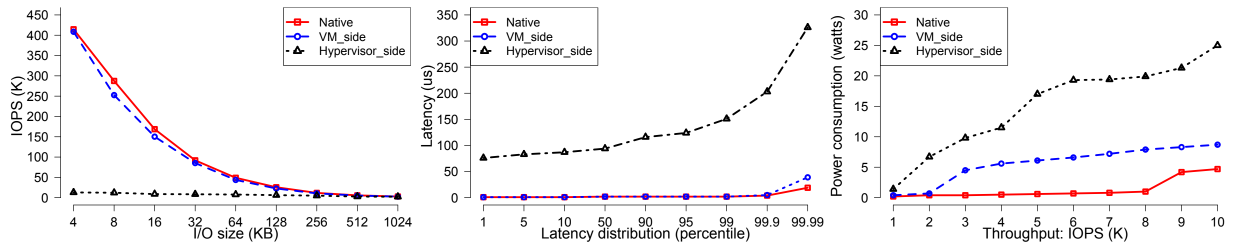
For efficient emulation of block devices, paravirtualization is the standard solution. Two of the most widely used para-virtualized device drivers are *virtio* [7] and *Xen paravirtualization* [10]. The former is widely used in *QEMU/KVM* based virtualization platforms; the latter is from the *Xen* project. *Xen PV* and *virtio* are architecturally similar, we discuss the *virtio* based storage stack in details. As it's shown in Figure 1, assume a guest OS issued a *read* request on some disk file, the activities of the guest OS and the host OS components are as follows:

1. The *read* request activates a Virtual Filesystem (VFS) function, passing to it a file descriptor and an offset.
2. If the request doesn't indicate direct I/O, the VFS function determines whether the required data are available in the page cache *1a*. If *1a* hits, the data are returned from the cache and the request is completed.
3. Assuming the page cache *1a* missed, the guest OS kernel must read the data from the block device. If the requested data reside in the flash cache, it's a flash cache hit at *1b*. In this case, data are fetched from the flash device and HDD access is obviated.
4. Assuming it's a flash cache miss, the request has to go through the traditional generic block and I/O scheduler layer and then be served by the virtual I/O device driver such as *virtio-blk*.
5. Upon a read request, the *virtio-blk* frontend driver composes a request entry and places it into the descriptor table of the *virtqueue*. Then, the *virtio-blk* frontend driver will call *virtqueue_kick*, which causes guest I/O exit and triggers a hardware register access called *VIRTIO_PCI_QUEUE_NOTIFY*.
6. *vhost* is the host-side *virtio* component for completing the virtual I/O request. Once *vhost* is notified by *KVM* for the guest kick, it fetches the *virtio* request from the queue and calls *QEMU*, which works as a regular userspace process, to complete the data transfer.
7. To fetch the data, *QEMU* issues I/O requests which again traverse the host OS storage stack. Host-side *page cache* or the optional *flash cache* are successively checked. Once the data hit at the caches or have been fetched from the HDD, *vhost* updates the *status* bit of the *virtio* request and issues an *irqfd* interrupt to notify the guest that the request is completed.

### 2.2 I/O Virtualization Overheads

In the virtual I/O path, there are two operations expensive in CPU cycles or request latency. The first is virtual I/O emulation, which requires intensive interactions between the *virtio* frontend and backend. Emulation causes frequent I/O interrupts and guest I/O exits, which are expensive in CPU cycles, as well as increase I/O latency. The second is the relatively slow HDD-based storage access, which costs milliseconds and has long been the bottleneck of cloud applications.

Both VM-side and hypervisor-side cache hits avoid



(a) Maximum *randread* throughputs with various I/O sizes. (b) Latency distributions of 4KB *randread* at the IOPS of 5000. (c) Power consumption of 4KB *randread* at various IOPS.

Figure 2: *fio* benchmark on DRAM-based Caches. *Native* denotes *fio* directly runs on the host machine and hits the host OS page cache; *VM_side* denotes *fio* runs on the VM and hits the guest OS page cache; *Hypervisor_side* denotes *fio* runs on the VM, misses the guest OS page cache but hits the host OS page cache.

the HDD access, thus, obviate the HDD latency. For hypervisor-side caching, *virtio* and *qemu* are always in the I/O critical path, thus, I/O virtualization overheads are inevitable. In contrast, for VM-side caching, if it's DRAM-based cache, *virtio* and *qemu* are not involved in the I/O path, because the DRAM of a VM is managed by *KVM* instead of *qemu*. *KVM* is much more efficient than *qemu* userspace emulation, especially with the support of hardware-assisted virtualization. If the cache is PCIe SSD-based, and the cache device is allocated to the VM via *PCI passthrough*, the I/O virtualization layers are also obviated. *PCI passthrough* is supported by *IOMMU*, which enables direct remapping of the guest physical address to host physical address, thus avoids I/O virtualization layers to apply the translations and obviates the I/O operation delay. However, if it's SATA SSD-based, even if the cache is logically VM-side, since the access to the cache device needs the involvement of *virtio* and *qemu*, I/O virtualization overheads still exist. As a result, VM-side caching is not superior to hypervisor-side caching for SATA SSD devices. To understand the I/O virtualization penalties as well as the performance and energy efficiency characteristics of various cache deployments, we quantitatively compare different cache schemes. Insights from the evaluation can direct future cache designs and optimizations of virtualization systems.

## 3 Evaluation of Cache Schemes

### 3.1 Evaluation Setup

**Cache Devices.** We use 4x2GB DDR2-800Mhz devices as the DRAM cache, one *Samsung 850 Pro* SATA SSD, and one 240GB *OCZ RevoDrive 3* PCIe SSD as flash caches.

**Measurement and Characterization Tools**. We use *fio* [11] as the I/O benchmark. *fio* enables various I/O workloads with optional parameters including *read/write* type, *sequential/random* access, I/O size, IOPS, and *O_DIRECT* etc.. We run *fio* on VMs. Setting the *direct* parameter enables I/O requests bypassing VM-side caches and hitting hypervisor-side caches. *fio* reports benchmark performance such as IOPS and latency distribution. We measure the power of the whole machine, because the cache access in virtualization platform involves intensive activities of multiple resources, including cache devices and CPUs. A *Watts Up? Pro ES* meter is used to measure the wall power of the machine.

**Experimental System**. Our system is equipped with an AMD Phenom II X4 B95 Quad-core 3.0 GHz processor with AMD-V virtualization support. The host OS is a 64-bit Ubuntu 15.04 with Linux kernel version 3.19.0-30-generic. QEMU emulator version 2.4.1 and KVM are used as the hypervisor. An official Ubuntu 15.04 64-bit Server Cloud Image is run on the VM as the guest operating system with 2 VCPUs and 2GB memory.

### 3.2 Evaluation Results

We choose *random read* as the I/O pattern of the *fio* benchmark to minimize the interference caused by potential data prefetching of operating systems. For each cache setting, we report the maximum throughput, latency distributions, and energy consumption of the benchmark. For latency distribution, we focus on the I/O size of 4KB, which is the default page management unit of most Linux operating systems.

***Observation 1 (on DRAM cache):***

The performance of VM-side caches is very close to the native caches in throughput and per I/O response time; while the hypervisor-side caches have an up to 97% performance penalty.

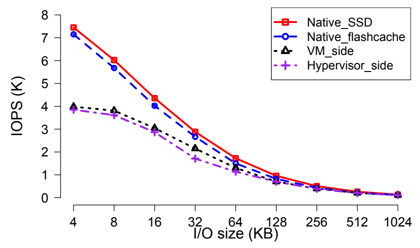
***Observation 2 (on DRAM cache):***

For 4KB small I/O requests, the maximum throughput of hypervisor-side caches is only about 3% of the VM-side caches; for 1MB large I/O requests, the maximum throughput of hypervisor-side caches is nearly the same as the VM-side caches.

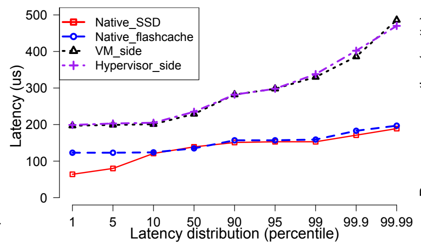
***Observation 3 (on DRAM cache):***

For same I/O throughput, hypervisor-side caches consume about 3x the power of VM-side caches.

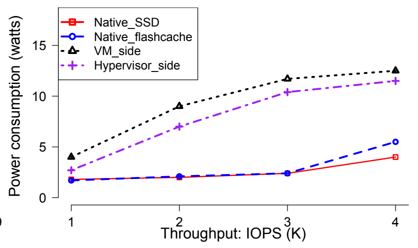
As it's shown in Figure 2(a), with various I/O sizes, the VM-side cache consistently achieves near-native



(a) Maximum *randread* throughputs with various I/O sizes.



(b) Latency distributions of 4KB *randread* at the IOPS of 3000.



(c) Power consumption of 4KB *randread* at various IOPS.

Figure 3: *fio* benchmark on SATA SSD-based Caches. *Native_SSD* denotes *fio* directly runs on the raw SSD of the host machine; *Native_flashcache* denotes *fio* directly runs on the host machine and hits the host OS *flashcache*; *VM_side* denotes *fio* runs on the VM and hits the guest OS *flashcache*; *Hypervisor_side* denotes *fio* runs on the VM, misses the guest OS *flashcache* but hits the host OS *flashcache*.

Table 1: *perf* system-event statistics during a 30-second cache access period with an IOPS of 5k. We ensure the cache hit at VM-side and hypervisor-side, respectively. *fio* is running on the VM; *perf* is running on the host OS to probe the system events caused by the VM process. For a direct comparison, the percentages of system events are normalized to the total number of hypervisor-side events. The units of *Count* and *Percent* are *million* and %, respectively.

| Event Source | Cache hit location | | | |
|---|---|---|---|---|
| | Hypervisor | | VM | |
| | Count | Percent | Count | Percent |
| kernel | 17545 | 60.90 | 1569 | 5.45 |
| qemu | 3295 | 11.44 | 0 | 0.00 |
| kvm | 2714 | 9.42 | 1478 | 5.13 |
| kvm_amd | 1811 | 6.29 | 1176 | 4.08 |
| libglib | 1220 | 4.23 | 0 | 0.00 |
| libpthread | 936 | 3.25 | 0 | 0.00 |
| vdso | 675 | 2.35 | 0 | 0.00 |
| libc | 609 | 2.12 | 0 | 0.00 |
| **Total** | **28809** | **100** | **4225** | **14.66** |

performance with a gap of less than 15%. In contrast, the hypervisor-side cache has a performance penalty of up to 97%. In Figure 2(b), the VM-side cache consistently achieves a near-native per I/O response time. In contrast, the hypervisor-side cache has a response time penalty of nearly 65 $\mu$s for a single 4KB read request. In Figure 2(c), for a same I/O throughput, the hypervisor-side cache consumes up to 3x the power of VM-side cache.

For DRAM, VM-side caches perform better and consumes less power than hypervisor-side caches. We believe the main reason is that the VM-side cache hit bypasses the *I/O virtualization layer*. Memory virtualization is implemented in the *KVM* kernel module, which is efficient with the support of hardware-assisted virtualization techniques such as *Intel VT-x* and *AMD-V*. In contrast, the I/O virtualization, including virtual I/O operations and disk emulation, is mainly managed by *QEMU*, which is a userspace process. The execution of virtual I/O requires frequent CPU mode switches, such as switches between *user* and *kernel* as well as *kernel* to *guest* mode, which are expensive in CPU cycles. As it's shown in Figure 1, when cache hits at *1a* (VM-side DRAM cache), virtual I/O and disk emulation are bypassed, the disk I/O operation is actually transformed to virtual memory access which is managed by *KVM*. When cache hits at *2a* (hypervisor-side DRAM cache), it implies a cache miss at *1a*, although the disk access can be avoided, the virtual I/O and disk emulation operations are still involved, thus, longer response time is observed by applications running on the VM, as well as a higher system power consumption.

To further verify our explanation, we conduct system event statistics during cache access under different caching schemes. *fio* benchmark is running inside a VM, and *perf* utility is employed to monitor the system events caused by the VM process. The *perf* statistic results are shown in Table 1. Generally, for a same amount of I/O requests, in hypervisor-side cache scheme, the total number of VM caused system events is 6x of the VM-side cache scheme. Specifically, in VM-side cache scheme, there are few userspace events caused, while, in hypervisor-side cache scheme, there are considerable user space system events such as events caused by *qemu* process and *GLib* library, etc.. This statistical analysis explains why hypervisor-side cache access is more costly than VM-side cache access.

From Figure 2(a) we can observe that as the I/O size increases, the throughput gap between *VM_side* and *hypervisor_side* cache schemes narrows down. We believe the reason is that for virtual I/O requests, the communication time between the *front-end (VM)* and the *back-end (Hypervisor)* is almost constant, thus for small requests the response time is dominated by the virtual I/O *round trip time (RTT)* between the VM and the hypervisor. When the request size increases, the real data transfer time dominates and the *RTT* becomes ignorable, thus, the throughput gap between *VM_side* and *hypervisor_side* cache narrows down.

Differing from Figure 2(a) and Figure 2(b) in which the *VM_side* and *Native* lines are almost overlapping, in Figure 2(c) there is an obvious gap between the *VM_side* and *Native* lines. The reason is that although VM-side memory access has a close-to-native performance, memory virtualization involves intensive activities of *KVM* module, which consumes extra power compared with native memory access.

***Observation 4 (on SATA SSD cache):***

The VM-side cache and hypervisor-side cache have similar performance in throughput and per I/O response time; both VM-side and hypervisor-side cache have an up to 60% performance penalty compared with the native SSD device or *flashcache*.

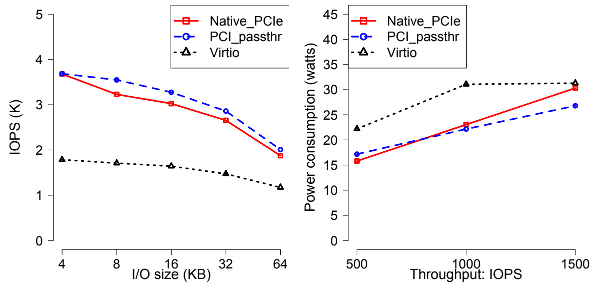
***Observation 5 (on SATA SSD cache):***

For 4KB small I/O requests, the maximum throughput of both VM-side and hypervisor-side caches is about 60% of the native SSD device or *flashcache*; for 1MB large I/O requests, the maximum throughput of VM-side and hypervisor-side caches is nearly the same as the native SSD device or *flashcache*.

***Observation 6 (on SATA SSD cache):***

For same I/O throughput, VM-side cache or hypervisor-side cache consumes up to 4x the power of the native SSD device or *flashcache* access.

Specifically, as it's shown in Figure 3(a) and Figure 3(b), in a wide range of I/O sizes, the hypervisor-side cache consistently delivers almost the same performance as the VM-side cache in throughput and per I/O response time. This observation differs from the observation made in the DRAM-based cache that VM-side caching has a much higher performance than hypervisor-side caching. We believe the main reason is that when the cache device is a SATA SSD, either the device is allocated to a VM to build a VM direct cache, or used by the hypervisor to build a hypervisor-side cache, a cache hit needs the involvement of *virtio* and *qemu*, thus, the applications on the VM observe similar response time. As it's shown in Figure 1, VM-side flash cache is denoted as *1b*, and hypervisor-side flash cache is denoted as *2b*. Although logically a cache hit at *1b* has a shorter access path, since the SATA SSD device cannot be accessed by VMs in the hypervisor passthrough way, the real data transfer still requires the involvement of the *virtio* driver. Thus, a cache hit at *1b* has similar overheads as a cache hit at *2b*. Since *virtio* is involved in either case, in Figure 3(a) we observe that either the hypervisor-side cache or the VM-side cache can only achieve about 60% of the native SSD I/O throughput for small requests. In Figure 3(c), we observe that for the same throughput both the hypervisor-side cache and the VM-side cache consume about 4x the power of native SSD access. To our surprise, the VM-side cache even consumes a little bit higher power than the hypervisor-side cache. We use *flashcache*, which is built on top of the Linux kernel's device mapper, as our cache implementation. Since we observe in Figure 3(c) that *Native_flashcache* has a



(a) Maximum *randread* throughputs with various I/O sizes.

(b) Power consumption of conducting 4KB *randread* at various IOPS.

Figure 4: *PCI passthrough* vs. *virtio*.

similar power consumption as *Native_SSD* access, we believe the extra power consumed in VM-side caching is not caused by the *flashcache* implementation, but virtualization components. In the case of VM-side caching, *flashcache* is built in the guest OS, and frequent guest OS activities cause considerable *KVM* module events. In contrast, in the case of hypervisor-side caching, *flashcache* is built in the host OS, cache activities will not stress the *KVM* module, causing less system events and consuming lower power.

***Observation 7 (on PCIe SSD cache):***[1]

The SSD allocated to the VM via *PCI passthrough* delivers near-native performance and energy efficiency.

A PCIe SSD is attached to a VM via *virtio* and *PCI passthrough*, respectively. We compare the performance and energy consumption of the SSD access in these two cases. As it's shown in Figure 4(a) and 4(b), *PCI passthrough* delivers near-native performance and energy efficiency. In contrast, *virtio* has obvious penalties in either throughput or system energy efficiency. This is coincident with our previous observations. To our surprise, *PCI passthrough* performs slightly better and consumes less power than native PCIe access. Since we currently employ a black-box method to analyze our test results, a further explanation to this surprising observation is our future work.

## 4 Potential Optimizations

Based on our tests, we believe the following considerations are practical for the design and implementation of virtualization systems in terms of cache performance and energy efficiency.

**Allocating DRAM resources directly to VMs.** Dynamic live VM memory allocation has been supported by *Xen* and *KVM*. Since VM-side caching delivers higher performance and energy efficiency than hypervisor-side caching, instead of using spare memory resource to build hypervisor-side caches [1, 2], memory resources can be allocated directly to VMs with high storage pressure. But

[1]This group of tests is conducted on another HP ProLiant DL370 G6 server for its IOMMU support.

for public clouds, dynamic live memory allocation requires the changes of pricing policies which need further investigations. Although hypervisor-side caches can build a distributed global cache pools to support VM with large working sets, we believe reclaiming the memory resource of idle VMs and VM migration are practical to meet the big-cache requirement.

**VM-side Block Device Read-ahead.** Even in the case that caches are built at the hypervisor side (*virtio* backend), the *virtio* frontend bulk prefetching can mitigate the communication overheads between the frontend and backend. Our tests show that for large requests, instead of the communication overheads, data transfer will dominate the response time. Thus, for workloads with intensive small requests, front-end bulk prefetching enables the communication overheads to be amortized. For instance, *block device read-ahead* of the guest OS can be set to prefetch data upon each virtual I/O request. The result is that even the I/O request size is 4KB, a larger bulk of data such as 128KB can be read ahead into the VM-side cache. This will benefit subsequent read requests targeting prefetched pages. A similar method has been employed by network file systems such as *NFS*, in which the default block size is 1MB, instead of 4KB, so as to reduce the network communication frequency between clients and servers.

**PCI Passthrough.** *PCI passthrough* is a practical way to allocate PCIe SSDs directly to VMs, so as to avoid the overheads of virtual I/O and disk emulation. However, *PCI passthrough* is limited to devices with PCI interfaces. SATA SSDs currently cannot be allocated in this way. This limits the deployment scope of *PCI passthrough*. Moreover, the exclusiveness of *PCI passthrough* limits each device being solely used by a single VM. A single PCIe SSD consumes a relatively high power of up to 20 watts. All these factors cripple the benefits of employing *PCI passthrough* for cache performance and energy efficiency purposes.

**Reducing Virtual I/O Overheads.** Linux kernel community continuously optimizes the *virtio* drivers. Virtio-blk [12], Virtio-blk-data-plane [13], and Virtio-blk Multi-queue [14] have been successively implemented to improve the *virtio* performance. DID [6] was proposed to reduce the I/O virtualization caused interrupt delivery overheads. Future congeneric optimizations on I/O virtualization will similarly benefit the performance and energy efficiency of hypervisor-side caches. Finally, container-based operating system level virtualization solutions such as *docker* can eliminate the overheads of hypervisor-based virtualization, thus, can be considered as an alternative in some cases.

## 5 Conclusion

This paper evaluates the impact of cache locations on storage performance and energy consumption of *QEMU/KVM* based virtualization systems. We present the performance and energy consumption results of VM-side and hypervisor-side caching using DRAM and SSD devices. Tests show that for DRAM-based caches with 4KB read requests, comparing with VM-side caching, hypervisor-side caching consumes 3x power while only achieves 3% of its throughput. We also demonstrate that I/O virtualization overheads are the culprit of hypervisor-side caching penalties. For SATA SSD-based caches, VM-side caching doesn't have any performance or energy efficiency superiority, because VM-side SSD caching also requires the involvement of *virtio*. Our tests show that *PCI passthrough* can bypass *virtio*, thus, eliminate the I/O virtualization overheads. Finally, we propose possible choices which will be useful for building high-performance and energy-efficient virtualization systems. Using the insights of this paper to optimize practical systems is our future work.

## References

[1] Infinio. (2014) Choosing a server-side cache: Why architecture matters. [Online]. Available: http://www.infinio.com/sites/default/files/resources/Infinio-technical-brief-architecture-matters.pdf

[2] J. Hwang, A. Uppal, T. Wood, and H. H. Huang, "Mortar: Filling the gaps in data center memory," in *VEE'14*, Salt Lake City, USA, March 2014.

[3] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer, "Mercury: Host-side flash caching for the data center," in *MSST'12*, Pacific Grove, USA, April 2012.

[4] T. Luo, S. Ma, R. Lee, X. Zhang, D. Liu, and L. Zhou, "S-cave: Effective ssd caching to improve virtual machine storage performance," in *PACT'13*, London, England, Septemebr 2013.

[5] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Geiger: Monitoring the buffer cache in a virtual machine environment," in *ASPLOS'06*, San Jose, USA, October 2006.

[6] C.-C. Tu, M. Ferdman, C. tung Lee, and T. cker Chiueh, "A comprehensive implementation and evaluation of direct interrupt delivery," in *VEE'15*, Istanbul, Turkey, March 2015.

[7] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, 2008.

[8] R. Shea, H. Wang, and J. Liu, "Power consumption of virtual machines with network transactions: Measurement and improvements," in *INFOCOM'14*, Toronto, Canada, April 2014.

[9] A. Kivity, Y. Kamay, F. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *OLS'07*, Ottawa, Canada, June 2007.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, , A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *SOSP'03*, New York, USA, October 2003.

[11] fio(1) - linux man page. Jens Axboe and Aaron Carroll. [Online]. Available: http://linux.die.net/man/1/fio

[12] A. He. (2012) Virtio-blk performance improvement. [Online]. Available: http://www.linux-kvm.org/images/f/f9/2012-forum-virtio-blk-performance-improvement.pdf

[13] K. Huynh and A. Theurer. (2013) Kvm virtualized i/o performance. [Online]. Available: http://www.novell.com/docrep/2013/05/kvm_virtualized_io_performance.pdf

[14] M. Lei. (2014) Virtio-blk multi-queue conversion and qemu optimization. [Online]. Available: http://www.linux-kvm.org/images/6/63/02x06a-VirtioBlk.pdf