Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.12785/amis/092L35

A Paravirtualized File System for Accelerating File I/O

*Kihong Lee, Dongwoo Lee, Dong Hyun Kang and Young Ik Eom**

College of Information and Communication Engineering, Sungkyunkwan University, Suwon 440-746, Republic of Korea

Received: 19 Jun. 2014, Revised: 23 Aug. 2014, Accepted: 24 Aug. 2014
Published online: 1 Apr. 2015

Abstract: Recently, virtualization technologies have gained widespread use in various systems since several new technologies assist virtual machines (VMs) in achieving high performance. However, I/O-intensive workloads still suffer from performance degradation due to CPU mode switching and duplicated I/O stacks in both the guest and host operating systems. In this paper, we propose a framework for improving file I/O performance in a virtualized environment, which consists of a paravirtualized file system, a shared queue, and an I/O-dedicated thread. The key ideas are to handle file I/O requests without the interference of the hypervisor and to have I/O requests bypass the guest I/O stack. To verify the performance improvement of our approach, we implemented a prototype and measured the performance. Experimental results demonstrate that our framework outperforms virtio, the predominant I/O virtualization framework, by providing 1.2–1.6x better throughput and that it eliminates most *vmexit*s during I/O process.

Keywords: I/O virtualization, paravirtualization, hypervisor, file system, I/O stack

1 Introduction

As a result of the development of high-performance virtual machine (VM) technologies, virtualization is now being widely used in both desktop and server systems. The methods to support virtualization can be categorized into hardware-based and software-based approaches. A hardware-based approach supports virtualization by adding special features to the CPU, memory, I/O devices, or other hardware components. With this, a VM can directly access physical devices or the device itself can recognize whether the tasks are being performed in a VM or not. On the other hand, a software-based approach can support virtualization without any special hardware features, but it sometimes needs a change in the guest kernel to improve performance or to assist with special virtualization functionality. Nowadays, while the hardware-based approach is dominant in processor and memory virtualization, both hardware- and software-based approaches are being actively studied for I/O device virtualization.

SR-IOV [1] and IOMMU [2] are well-known techniques for hardware-based I/O virtualization. By providing the direct access interface to the VM, these techniques provide high performance close to that of non-virtualized systems. However, there are only a few products that have adopted the techniques because they increase the cost of the product. Additionally, cloud service centers are unwilling to adopt the hardware-based I/O virtualization technology because it may weaken key characteristics of virtualization, such as portability, flexibility, and security since it allows each VM to control the physical devices directly. In contrast, a software-based approach makes it easy to encapsulate VMs (e.g., for live migration or VM snapshot creation) and therefore it can more easily facilitate virtualization. Software features can also enable a VM to utilize virtual devices with no physical counterpart (e.g., a image file can be used as a block device). On account of these advantages, software-based techniques are currently preferred for I/O device virtualization such as virtio [3] and split driver model implemented in Xen [4]. Unfortunately, a software-based approach provides relatively worse performance than hardware-based one because the entire virtualization mechanism should be done only with software.

In this paper, we concentrate on two major causes of performance degradation in software-based I/O virtualization. First, the hypervisor should interfere with the guest I/O process because a VM has no privilege to access physical I/O devices. Thus, whenever a VM executes an I/O operation, the CPU core running the VM should switch its mode to host mode, i.e. *vmexit*, to handle it. Second, there exist duplicated I/O stacks during the process of I/O requests. In a hosted virtual machine

* Corresponding author e-mail: yieom@skku.edu

© 2015 NSP
Natural Sciences Publishing Cor.

environment, a host and a guest have their own OS, each of which contains the I/O stack. Some previous studies pointed out the problem [5,6], and some parts of the duplicated I/O stack are redundant and therefore can be skipped.

In order to resolve these problems, we propose a new framework for a virtualized environment that remarkably improves file I/O performance. It consists of three components: a paravirtualized file system, a shared queue, and an I/O-dedicated thread. With the paravirtualized file system, I/O requests are sent to the host directly rather than to the block layer of the guest OS, and so we can avoid the execution of a redundant I/O stack on the guest side. The I/O-dedicated thread and the shared queue help VMs reducing the number of *vmexit*s by utilizing a polling mechanism.

To give details of our framework, this paper is organized as follows. In Section 2, we give an overview of related work. Section 3 describes our framework and its components in detail. The experimental results of our prototype are presented in Section 4, and Section 5 discusses some issues related to our approach, including compatibility and safety, and also introduces alternatives for the limitations of our framework. Finally, we conclude this paper in Section 6.

2 Related Work

One of the predominant device driver frameworks for virtualization is virtio [3], which has been included in the mainline of the Linux kernel since version 2.6.24. Virtio utilizes a shared queue called virtqueue so the guest device driver sends I/O requests to the queue rather than directly executes the I/O operation. Since full emulation of I/O devices is complicated and inefficient, virtio relatively achieves better performance than typical device emulation. When numerous requests are generated simultaneously, virtio is able to coalesce the requests to reduce the number of *vmexit*s. To handle the requests on the host side, however, *vmexit* should be triggered and it causes significant overheads. In particular, the faster a device operates, the more *vmexit*s are generated in given time. This implies that virtio cannot control *vmexit*s well, whereas our proposed approach remarkably reduces the number of *vmexit*s by using a paravirtualized file system and a polling mechanism.

Some previous studies have suggested that VM performance can be improved through the structures using dedicated CPU cores. SplitX [7] is a new virtualization model in which the hypervisor runs on the dedicated CPU core. The suggested model can reduce the direct and indirect costs associated with *vmexit*s. On the contrary, our approach uses an I/O-dedicated thread to handle block I/O requests. VPE [8] and ELVIS [9] are systems using dedicated cores to efficiently handle I/O requests. Their studies demonstrated that using dedicated cores for I/O is an effective way to reduce *vmexit*s in a virtualized environment. These two studies aimed to improve the performance of general I/O devices and modified the device driver to add virtualization awareness. On the other hand, we only focus on the block I/O and file system layers. In file I/O process, our approach not only effectively reduces *vmexit*s but also bypasses the redundant block I/O stack on the guest side, so we can achieve better performance enhancement in file I/O than them.

There is some research on adding virtualization awareness to the components of file system layer. VirtFS [10] was proposed as a virtualization-aware file system. It uses a client-server model similar to that of network file systems and utilizes virtio framework. Both VirtFS and our approach follow the same concept of optimizing I/O procedures by utilizing virtualization awareness at the file system level, but we further take into account the overhead caused by *vmexit*s. Multilanes [11] is a storage system for OS-level virtualization on many-core systems. It has a similar mechanism as ours with respect to directly delivering guest block I/O requests to the host. However, Multilines focuses on the contention of shared data structures between multiple guest systems, and eliminates performance interference by partitioning the VFS and the I/O device driver.

3 Architecture Design

In this section, we explain the structure and mechanism of the proposed framework. Frequently generated *vmexit*s and duplicated I/O stacks are major causes of performance degradation in virtualized environments. Since the overheads generated by *vmexit* makes up a significant portion, minimizing the occurrence of *vmexit*s is one of the best ways to improve the system performance [12]. Accordingly, we suggest a paravirtualized I/O framework that enables a VM to interact with the hypervisor without *vmexit*s. Another virtue of the design is the elimination of the redundant guest I/O stack. In a hosted virtual machine environment, there exist multiple I/O stacks. One is for the host OS and the others are for the VMs. Therefore, a request will pass through two block I/O stacks but some procedures within the I/O stack do not need to be performed twice. Our proposed framework consists of a paravirtualized file system, a shared queue, and an I/O-dedicated thread. The overall system structure is illustrated in Figure 1.

3.1 Paravirtualized File System

Paravirtualized file system is aware of virtualization so it can behave differently from typical file systems. The main role of the paravirtualized file system is to bypass the redundant guest I/O stack. When an application requests file I/O, the paravirtualized file system passes the I/O

© 2015 NSP
Natural Sciences Publishing Cor.

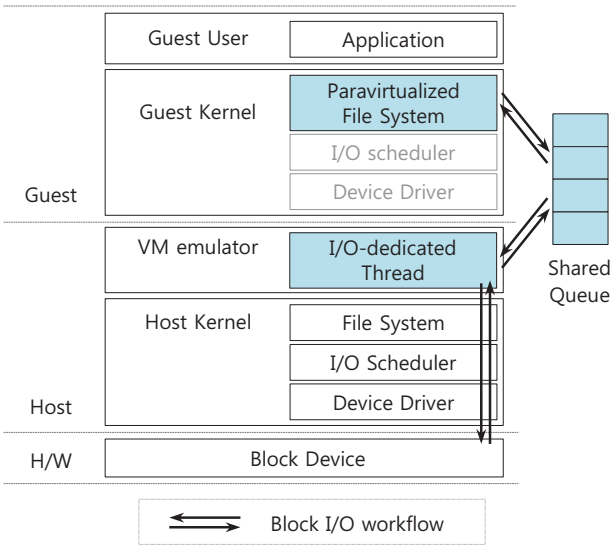

Fig. 1: The structure of a proposed I/O framework

requests to the host instead of to the block layer of guest kernel. After the requests are processed in the host, the file system takes the outcome of the request from shared queue, and acts the same as if it was handled in the block layer of the guest.

A file system is the first step in handling file I/O requested by user applications. If I/O requests are passed to the host at this point, the rest of the guest I/O stack, e.g., I/O scheduler and device driver, can be bypassed. Compared to the I/O process in a non-virtualized system, duplicated I/O stacks are one major cause of performance degradation. In particular, when a VM utilizes image files as block devices, merging and sorting operations of an I/O scheduler may have negative effects on the I/O process in the host. This is because consecutive disk blocks from a viewpoint of the guest are no longer consecutive in the host, and it depends on the file system type of both the host and the guest as well as the format of the disk image file. Consequently, eliminating a redundant I/O stack of the VM can improve file I/O performance.

Bypassing guest I/O stack also enables the VM to avoid caching files. In a hosted virtualization environment, files are cached in memory of both the guest and the host whenever the files are accessed in a VM, and it incurs waste of memory. Moreover, it becomes serious when multiple VMs access a single file concurrently. Each guest OS caches the file separately and the host OS regards each instance of the file as a different one. As a result, one file may be loaded into memory twice the number of VMs. The main reason of this situation is the lack of information sharing between the host and the guest. There are several studies on reducing data duplication in page cache and they propose to provide the host with additional information on the pages accessed by the VM in order to reduce the duplication [13,14]. Our paravirtualized file system also can deliver semantic information on the file system layer to the host OS to reduce page duplication, as well as it can avoid file caching in VMs.

3.2 Shared Queue Between Host and Guest

The shared queue is used to support interaction between the guest and the host. Originally an I/O operation triggers a *vmexit* because it attempts to access a physical block device but the VM has no privilege for the access. When the shared queue is used, however, I/O operations are substituted with memory operations which can be executed in the VM directly. In this context, the memory address should be appropriately translated among the guest virtual addresses, the guest physical addresses, and the host virtual addresses.

After block I/O requests are enqueued from the paravirtualized file system, the I/O-dedicated thread handles them and updates the completion status in the queue to TRUE. Finally, the file system dequeues the requests having TRUE of completion status and continues the remaining I/O process. To carefully manage I/O requests generated by several VMs, each VM has its own shared queue. With this, the I/O-dedicated thread can select the queue of the VM to be handled preferentially. In order to share the queue, both the VM emulator and the virtual machine should know the memory address of the queue. In a typical hosted virtualization environment, it is not difficult to share memory addresses because the emulator and its virtual machine have the same address space.

3.3 I/O-dedicated Thread

Our framework makes the guest OS delegate the block I/O process to the I/O-dedicated thread, and through the shared queue I/O requests are sent to the host without triggering a *vmexit*. However, there remains one more problem: notifying the host of arriving the new requests in the queue. virtio also utilizes the shared queue called virtqueue so it can deliver the request to the host without a *vmexit* too. To notify, however, virtio intentionally raises a *vmexit* [3]. To avoid this we introduce an I/O-dedicated thread which monitors the queue to determine whether new requests have arrived or not. The thread utilizes a polling mechanism to check the queue as fast as possible. Although the polling thread negatively affect CPU usage, it can improve I/O performance when I/O-intensive workloads are running on the system. In addition, block I/O operations are asynchronous, and so one I/O-dedicated thread can simultaneously handle several I/O requests from many VMs.

© 2015 NSP
Natural Sciences Publishing Cor.

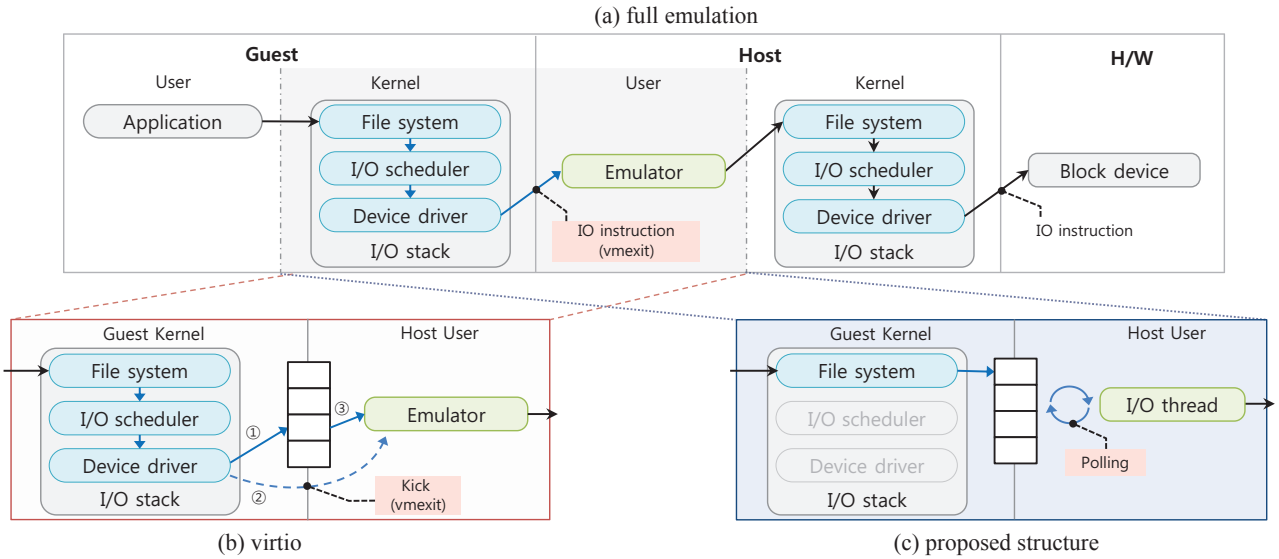

Fig. 2: Comparison of file I/O workflows in a virtualized environment

3.4 File I/O Workflow

In a virtualized environment, the rudimentary mechanism of virtualizing an I/O device is the device emulation in which the emulator provides a virtual device to the VM and emulates them with software. This is also called full emulation and its workflow is illustrated in Figure 2a. When a guest user application invokes a system call for file I/O, the file system receives and processes the request. The file system translates the file offset of the request into a corresponding block number and generates block I/O requests. Next, the requests are sorted and merged by the I/O scheduler and are sent to the device. At this point, the device driver attempts to execute I/O instructions, but a VM has no privilege to directly access physical devices. Thus, the CPU running as a vcpu immediately switches its mode to the host mode (i.e., *vmexit*) and the emulator handles the exceptional situation. The emulator typically runs in a host user mode, thereby the rest of the process is almost equal to that in the previous steps. The only difference is that the host has the privilege to control physical devices so I/O instructions can be executed directly. Namely, *vmexit* occurs whenever an I/O operation is executed, and several previous studies [12, 15] have already pointed out the performance degradation problem due to the overhead of executing I/O instructions in a VM environment.

Figure 2b presents the block I/O workflow of the system using virtio. The notable difference with full emulation is the delivery process of I/O requests between the guest and the host. Virtio utilizes virtqueue that is shared between the guest and the host. The virtio driver enqueues I/O requests to the shared queue and then raises *vmexit* through the kick operation. After that, the emulator handles the requests in the queue. In the case where numerous I/O requests are generated by I/O-intensive tasks, virtio can reduce the occurrence of *vmexit*s, by accumulating several requests in the queue and handling them together with a single kick operation.

In comparison, the I/O workflow of our proposed scheme can be described as seen in Figure 2c. It also utilizes a shared queue to support the interaction between the guest and the host, in a manner similar to virtio, but the actor is a file system. The paravirtualized file system is aware of virtualization and so sends requests to the shared queue instead of the block layer of the guest kernel. In addition, the I/O-dedicated thread actively checks whether new requests are put in the queue or not. Through these mechanisms, the proposed structure can bypass the guest block layer and can avoid triggering *vmexit*s.

4 Evaluation

In this section, we evaluate the proposed framework. To verify the performance improvement of our approach, we implemented a prototype and measured the file I/O throughput and latency by using some benchmarks. We also analyzed the overhead of *vmexit*s under our framework and compared it with that of virtio.

4.1 Prototype Implementation

We used FUSE [16], a framework for file systems in userspace, to handily manipulate file systems. We implemented our scheme using fuse-ext2 [17] which is an implementation of ext2 file system based on FUSE. The overall mechanism of our modified fuse-ext2 is the same

© 2015 NSP
Natural Sciences Publishing Cor.

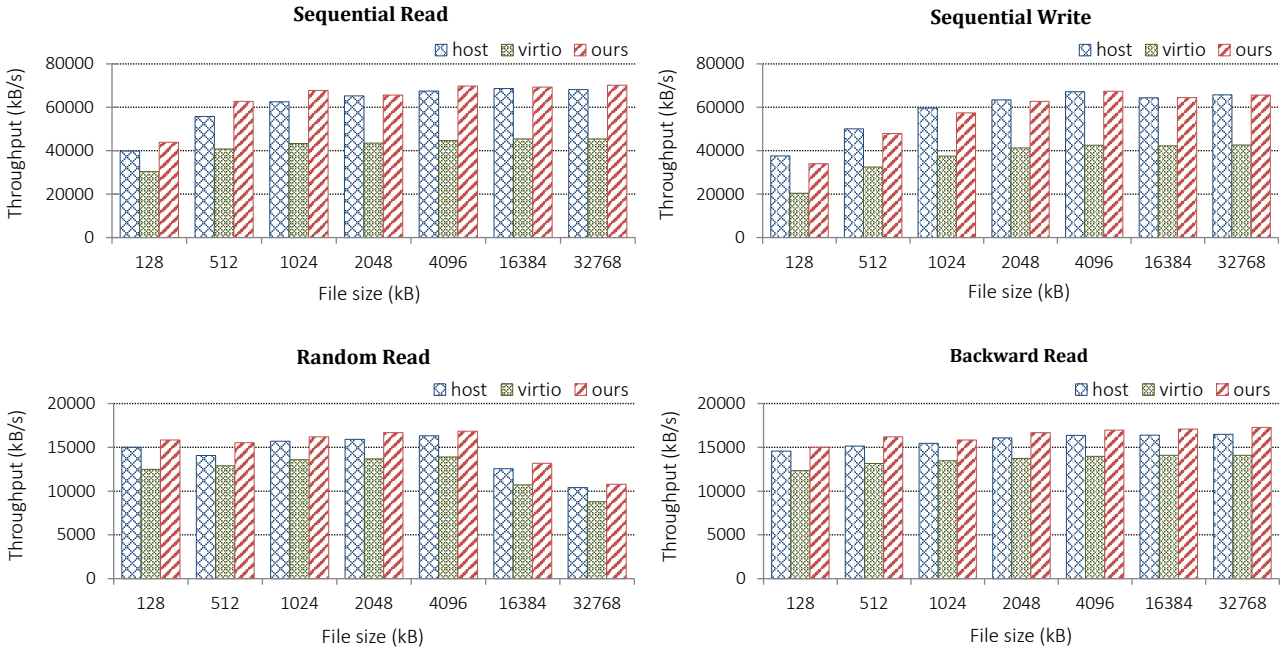

Fig. 3: File I/O performance of IOzone benchmark

as that of the original fuse-ext2, except the modified one sends I/O requests to the shared queue rather than to the block layer of the guest kernel at the end of the internal process. A file system implemented with FUSE, of course, has partial differences with the kernel-level file system, but it does not seem to be crucial just to verify the performance improvement from our approach. We used the Native Linux KVM tool [18] to emulate VMs, and added the shared queue and the I/O-dedicated thread to it. A non-blocking concurrent queue algorithm was applied to the shared queue to ensure consistency and scalability because the queue can be simultaneously accessed by the I/O-dedicated thread and applications of the VM.

4.2 Experimental Setup

The test machine was equipped with an Intel i5 3570 processor (4 cores, 3.4 GHz) and 4 GB of RAM. A Samsung 470 Series SSD was used for the experimental disk, and all programs including the OS were installed on other disks. We used KVM [19] as the hypervisor and assigned one virtual CPU, i.e., vcpu, and 2 GB of memory to the guest machine. The OSs of the host and the guest were Ubuntu 12.04 with Linux kernel version 3.9.0.

For fair experiments, we created three 16 GB partitions on the SSD. The first partition was used to test the file I/O of the host system, and the second and third partitions were respectively used for virtio and for our framework in the VM. Each VM and host system use raw partitioned disks. To avoid effects of page cache, assignment the disk to VM and all file I/O operations in the VM were executed with the `O_DIRECT` option. Meanwhile, the I/O-dedicated thread of our prototype uses polling mechanism while running on an dedicated CPU core in the host.

4.3 Throughput Analysis

We measured file I/O throughput in the native host (i.e., the non-virtualized system), in a guest using virtio, and in a guest using our framework. To compare our framework with the others, all file system environments were based on fuse-ext2. The host and the guest with virtio used an unmodified fuse-ext2, and the guest with our framework used the modified one mentioned above.

We utilized IOzone [20] benchmark to evaluate file read/write throughput of various patterns, such as sequential read/write, random read/write, and backward read. The benchmark read and wrote the files whose sizes were from 128 KB to 32 MB, and the results are describes in Figure 3. Ours achieves better throughput than virtio in all cases, and the throughput is almost the same as that of the native host system within the margin of error. The throughput of ours sometimes exceeds that of the native system because the I/O-dedicated thread of our framework handles I/O requests instead of the guest kernel. It can provide our framework with the advantage of high cache hit ratio and low context switching overhead. In the case of sequential read/write, our framework achieves 40%–60% better performance than virtio. Due to the direct I/O, which makes the block I/O bypass the page cache, the results of the write operations

© 2015 NSP
Natural Sciences Publishing Cor.

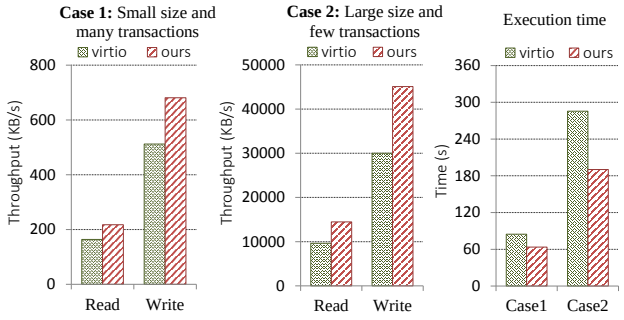

Fig. 4: File I/O performance measured by PostMark

are similar to those of the read operations. In the case of random read/write and backward read, our prototype outperforms virtio by 20%–25%. Random write also shows similar results to random read, so we omit those results. In a typical block device, non-sequential I/O is slower than sequential I/O, so random and backward I/O are less affected by the overhead generated by *vmexit*s. As a result, the performance of non-sequential operations shows less improvement in our prototype.

Figure 4 shows the results of random file I/O measured via Postmark. In Case 1, the benchmark randomly reads and writes files with sizes from 0.5 KB to 10 KB with 5,000 transactions. In Case 2, it performs the measurement with larger file sizes, from 1 MB to 20 MB with 500 transactions. Case 3 shows the execute time of the experiments. In both cases, our prototype has better performance than virtio. The execution time of ours relative to that of virtio is shorter by 25% for Case 1 and by 33% for Case 2. Postmark randomly accesses the files, but it sequentially reads or writes the blocks within a single file. in Case 2 where the file size is larger and the number of files are smaller, therefore the access pattern is more sequential than Case 1, and overall throughput of Case 2 gets better. Thus, similar to the result of IOzone, our prototype achieves higher performance improvements when blocks are more sequentially accessed and when the block device performs faster.

4.4 Latency Analysis

Latency is another important factor in I/O performance. Our framework can reduce I/O processes by eliminating the redundant I/O stack of the guest, and also shortens the latency time of each I/O operation by using the polling mechanism. To validate this, we measured file I/O latency with IOzone benchmark and the results are presented in Figure 5. The benchmark performs 4KB sequential read/write to a 32 MB file and measures the latency of each operation. In the case of virtio, the average read and write latencies were 94.64 µs and 91.31 µs, respectively. On the other hand, our prototype spent 63.05 µs and 61.56 µs on average for each read and write operation. Therefore, the I/O response time of ours was approximately 33% shorter than that of virtio.

There are some peak points far above the average value in all graphs, and their frequency is much higher than that of the non-virtualized system. These peaks are thought to be caused by virtualization. The major cause of huge delay is due to VM scheduling in the host, which can make the VM lose its vcpu.

4.5 Occurrence of vmexits

One advantage of our approach is that file I/O does not trigger *vmexit*s. To verify this, we further analyzed the pattern of *vmexit* occurrences in following cases:

(1) VM is in an idle state
(2) I/O-intensive workload is running with virtio
(3) The workload is running with our framework

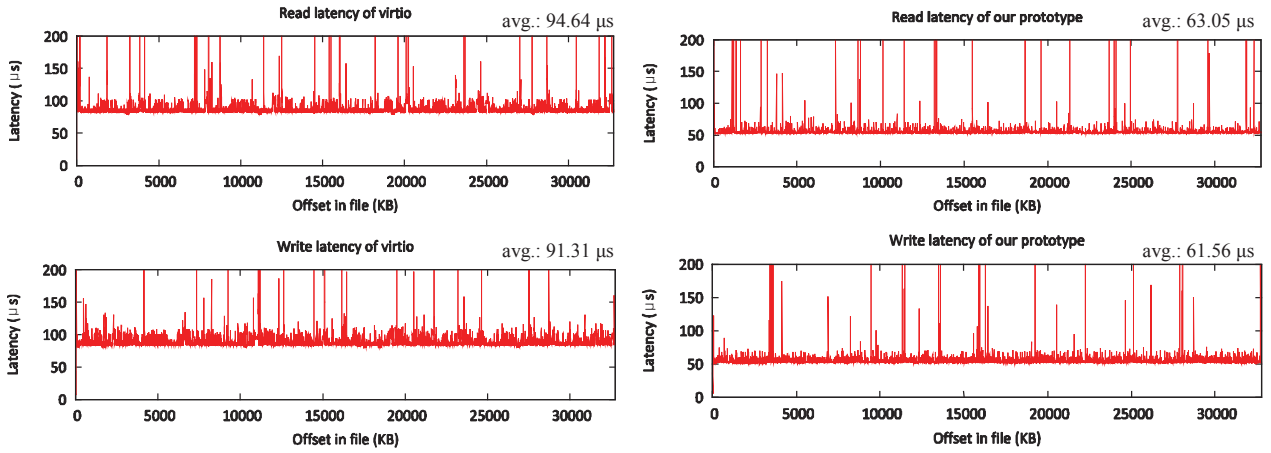

Fig. 5: File I/O latency measured by IOzone benchmark

© 2015 NSP
Natural Sciences Publishing Cor.

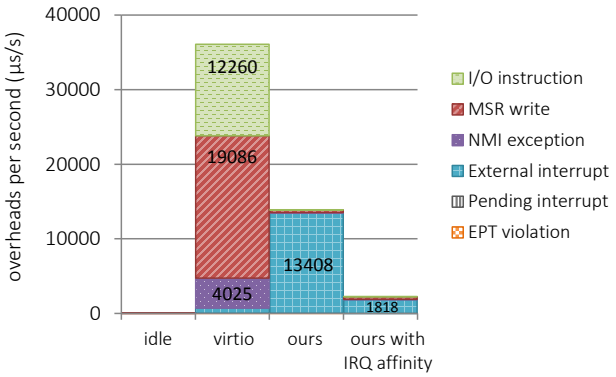

Fig. 6: Detailed overheads caused by *vmexit*s

(4) The workload is running with our framework and with an IRQ affinity setting

We used perf tools [21] to count the discrete events of *vmexit*s, and Figure 6 describes the results. In an idle state, there were almost no *vmexit*s, and the overhead caused by *vmexit*s was only 0.0055%. In the case of virtio, the overhead was of about 36 ms per second. In our framework, on the other hand, most of the I/O instructions, MSR writes, and NMI exceptions were disappeared. However, relative to the idle state, *vmexit*s consumed about 13.9 ms, and the overhead was mainly caused by external interrupts. This is due to hardware interrupts; when a physical core running on the VM receives hardware interrupts, it will switch to host mode to handle them. In the case of our framework, I/O requests are processed by the I/O-dedicated thread instead of the vcpu, so the physical core stays in the guest mode most of the time. To prevent the above problem, we set IRQ affinity to deliver the interrupts of block I/O to the I/O-dedicated thread. The interrupts from the block I/O device are thereby delivered to the I/O-dedicated thread instead of the VM, thus the amount of *vmexit*s can be significantly reduced. As a result, the number of external interrupts is reduced by 87%, and finally our framework saves about 34 ms per second against virtio. The remaining external interrupts are caused by the local timer, rescheduling, etc. These interrupts can also be removed with an IRQ affinity setting, but it is thought to be inordinate configuration.

5 Discussion

As previously mentioned, our proposed framework includes a paravirtualized file system, and this means that the guest OS should be modified. Therefore our framework cannot be applied to proprietary OSs which restrict such modifications, e.g., Microsoft Windows. However, Linux allows to build a new file system as a module, so the paravirtualized file system can be utilized without re-compiling the kernel. Additionally, if the file system of our framework is implemented based on a pre-existing file system, the two file systems can be fully compatible with each other. For example, if the paravirtualized file system is based on ext4, our framework can utilize intact block devices formatted with ext4. This is because the modification for paravirtualization only alters the destination of the block I/O requests, and it does not impact the intrinsic mechanisms of the file system.

We should address the security issues of our approach because it delegates a partial role of the guest to the host. However, originally a VM could not directly execute I/O operations, except in the case where SR-IOV devices are used. In all other cases, block I/O requests from a VM should be handled by the host. Our approach also delegates the role of block I/O process to the host. With the paravirtualized file system, only the point of delivery of the I/O requests is put forward, and this would not cause an additional tradeoff in safety or isolation.

Request coalescing is another consideration. Some previous studies for improving I/O performance suggested batching several I/O requests into one bundle [3,5,8]. In the case of virtio, a *vmexit* is triggered in order to notify the host of newly arrived requests, and it incurs a considerable delay. Coalescing can improve overall I/O throughput, but it may increase the latency of each request. The approach proposed in this paper, in contrast, does not involve a *vmexit* during the I/O process so the delay is minimized by sacrificing one core to poll the shared queue. Thus, our approach can achieve the high performance improvement close to that of non-virtualized systems without coalescing I/O requests, and it also does not sacrifice the file I/O latency.

In our proposed framework, the I/O-dedicated thread uses the polling mechanism to monitor the shared queue. With polling, we can avoid triggering *vmexit*s and reduce I/O latency. However, it is not suitable for every situation because the polling mechanism occupies one CPU core all the time. In mobile devices or embedded systems, there are restrictions on the number of cores, and polling also causes excessive power consumption. Besides, I/O-intensive tasks are not always running in VMs in many cases. In these cases, one alternative would be an event-driven mechanism. In this mechanism, I/O-dedicated thread wakes up and processes I/O requests only when receiving an event signals from the guest rather than always polling the shared queue. This may increase I/O latency against the polling mechanism but it can efficiently utilize the CPU in case I/O requests are generated infrequently. Therefore, dynamic switching between polling and event-driven mechanism considering the number of I/O requests and core utilization will help our proposed framework overcome its limitations and enhance adaptability in various situations.

© 2015 NSP
Natural Sciences Publishing Cor.

6 Conclusion

In a virtualized system, I/O-intensive workloads suffer from performance degradation when compared with a non-virtualized environment. First, access to physical I/O devices leads to *vmexit*s, and second, there are redundant I/O stacks because of the nested OS. In this paper, we proposed a new framework to improve file I/O performance by eliminating the two causes mentioned above. To deliver I/O requests without a *vmexit*, we used a shared queue and an I/O-dedicated thread. This thread uses a polling mechanism to promptly handle requests after they arrive in the shared queue. Our framework also adopted a paravirtualized file system that bypasses the redundant I/O stack of the guest when it receives I/O requests, achieving a further enhancement in file I/O performance. The polling mechanism makes the I/O thread occupy one CPU core all the time, but this will have little effect on the system because nowadays the number of cores on a single CPU has been increasing gradually. Therefore, our proposed approach can be persuasive if it provides a reasonable performance improvement. To verify this, we implemented a prototype and evaluated our approach. The results show that our proposed framework improves throughput by 20%–60% and reduces latency by 33% against virtio on average. Also, most of the *vmexit*s were eliminated. Moreover, if hardware itself achieves better performance, the frequency of the *vmexit*s will increase, and therefore our approach would achieve an even greater performance improvement.

Currently, we are investigating how to apply our approach to ext4, a kernel-level file system. This would help us to verify our approach more practically and accurately. In the future, we plan to study how to efficiently handle requests generated by numerous VMs.

Acknowledgement

This research was supported by the IT R&D program of MKE/KEIT (10041244, SmartTV 2.0 Software Platform) and Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2010-0020730).

References

[1] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, High Performance Network Virtualization with SR-IOV, Journal of Parallel and Distributed Computing **72**, 1471-1480 (2012).
[2] AMD, I/O Virtualization Technology (IOMMU) Specification, AMD Pub. (2009).
[3] R. Russell, Virtio: Towards a De-facto Standard for Virtual I/O Devices, ACM SIGOPS Operating Systems Review **42**, 95-103 (2008).
[4] D. Chisnall, The Denitive Guide to the Xen Hypervisor, Prentice Hall Open Source Software Development Series (2008).
[5] A. Gordon, M. Ben-Yehuda, D. Filimonov, and M. Danhan, VAMOS: Virtualization Aware Middleware, In Proceedings of the 3rd Workshop on I/O Virtualization (2011).
[6] J. Liu, W. Huang, B. Abali, and D. K. Panda, High Performance VMM-Bypass I/O in Virtual Machines, In Proceedings of the 2006 USENIX Annual Technical Conference, 29-42 (2006).
[7] A. Landau, M. Ben-Yehuda, and A. Gordon, SplitX: Split Guest/Hypervisor Execution on Multi-core, In Proceedings of the 3rd Workshop on I/O Virtualization (2011).
[8] J. Liu and B. Abali, Virtualization Polling Engine (VPE): Using Dedicated CPU Cores to Accelerate I/O Virtualization, In Proceedings of the 23rd International Conference on Supercomputing, 225-234 (2009).
[9] N. HarEl, A. Gordon, A. Landau, M. Ben-Yehuda, A. Traeger, and R. Ladelsky, Efficient and Scalable Paravirtual I/O System, In Proceedings of the 2009 USENIX Annual Technical Conference, 231-242 (2013).
[10] V. Jujjuri, E. Van Hensbergen, A. Liguori, and B. Pulavarty, VirtFS – A Virtualization Aware File System Pass-through, In Proceedings of the 2010 Linux Symposium, 109-120 (2010).
[11] J. Kang, B.Zhang, T.Wo, C. Hu, and J. Huai, MultiLanes: Providing Virtualized Storage for OS-level Virtualization on Many Cores, In Proceedings of the 12th USENIX Conference on File and Storage Technologies, 317-329 (2014).
[12] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon, Software Techniques for Avoiding Hardware Virtualization Exits, In Proceedings of the 2012 USENIX Annual Technical Conference, 373-385 (2012).
[13] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman, Satori: Enlightened Page Sharing, In Proceedings of the 2009 USENIX Annual Technical Conference, 1-14 (2009).
[14] N. Amit, D. Tsafrir and A. Schuster, VSWAPPER: A Memory Swapper for Virtualized Environments, In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, 349-366 (2014).
[15] 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 SIGARCH Computer Architecture News **40**, 411-422 (2012).
[16] M. Szeredi, FUSE: Filesystem in Userspace, http://fuse.sourceforge.net.
[17] fuse-ext2, http://alperakcan.net/projects/fuse-ext2.
[18] Native Linux KVM tool, https://github.com/penberg/linux-kvm.
[19] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, KVM: The Linux Virtual Machine Monitor, In Proceedings of the 2007 Linux Symposium, 225-230 (2007).
[20] W. Norcott, IOzone Filesystem Benchmark, http://www.iozone.org.
[21] A. Melo, The New Linux 'perf' Tools, In Proceedings of the 17th International Linux System Technology Conference (Linux Kongress), 21-24 (2010).

© 2015 NSP
Natural Sciences Publishing Cor.

Kihong Lee received the B.S. degree in College of Information and Communication Engineering from Sungkyunkwan University in 2013, and he is currently a M.S. student at Sungkyunkwan University. His research interests include virtualization, operating systems, and embedded systems.

Dongwoo Lee received his B.S. degree in the Department of Computer Engineering of Sungkyunkwan University, Korea in 2010 and M.S. degree in the Department of Mobile Systems Engineering from Sungkyunkwan University in 2012. He is currently a Ph.D. candidate in the Department of IT Convergence of Sungkyunkwan University. His current research interests include virtualization, cloud computing, and storage systems.

Dong Hyun Kang received the B.S. degree in Computer Engineering from Korea Polytechnic University, Korea, in 2007, and the M.S. degree in College of Information and Communication Engineering from Sungkyunkwan University, Korea, in 2010. He is currently a Ph.D. student at Sungkyunkwan University. His research interests include storage systems, operating systems, and embedded systems.

Young Ik Eom received his B.S., M.S., and Ph.D. degrees from the Department of Computer Science and Statistics of Seoul National University in Korea, in 1983, 1985, and 1991, respectively. He was also a visiting scholar in the Department of Information and Computer Science at the University of California, Irvine from Sep. 2000 to Aug. 2001. Since 1993, he is a professor at Sungkyunkwan University in Korea. His research interests include system software, operating system, virtualization, and system securities.

© 2015 NSP
Natural Sciences Publishing Cor.