Appears in the Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)

Adding Advanced Storage Controller Functionality via Low-Overhead Virtualization

Muli Ben-Yehuda, Michael Factor,
Eran Rom, and Avishay Traeger
IBM Research–Haifa
{muli,factor,eranr,avishay}@il.ibm.com

Eran Borovik and Ben-Ami Yassour
{eran.borovik,benami.yassour}@gmail.com

Abstract

Historically, storage controllers have been extended by integrating new code, e.g., file serving, database processing, deduplication, etc., into an existing base. This integration leads to complexity, co-dependency and instability of both the original and new functions. Hypervisors are a known mechanism to isolate different functions. However, to enable extending a storage controller by providing new functions in a virtual machine (VM), the virtualization overhead must be negligible, which is not the case in a straightforward implementation. This paper demonstrates a set of mechanisms and techniques that achieve near zero runtime performance overhead for using virtualization in the context of a storage system.

1 Introduction

Additional functions, such as file serving or database, are often added to existing storage systems to meet new requirements. Historically, this has been done via code integration, or by running the new function on a gateway or virtual storage appliance (VSA [37]). Code integration generally performs best. However, the new function must run on the same OS version, the controller's main functionality is vulnerable to bugs due to lack of isolation, resource management is complicated for software which assumes a dedicated system, and development complexity increases in particular when the new function already exists as independent software. The gateway approach offers isolation but adds both latency and hardware costs.

A hypervisor can isolate the new function, allow for differing OS versions, and simplify development. However, until now the high performance overhead of virtualization (in particular virtualized I/O) has made this approach impractical. In this paper, we show how to use server-based virtualization to integrate new functions into a storage system with near zero performance cost. Our approach is in line with the VSA approach, but we run the VM directly on the storage system.

While our work was done using KVM [14], our insights are not KVM-specific. We do take advantage of the fact that KVM uses an asymmetric model in which some of the code is virtualized (the new features) while other code (the original storage system) runs on "bare metal," unaware of the existence of the hypervisor.

There are three sources of performance overhead. *Base overheads* include aspects such as virtual memory management or process switching. *External communication* with storage clients is important when the new function is a "filter" on top of the original storage system, e.g., a file server. Finally, *internal communication* overheads are incurred to tie the new function to the original controller.

To reduce base overhead, we use two main techniques. First, we statically allocate CPU cores to the guest to ensure that the function has sufficient resources. Second, we statically allocate memory for the VM, backing that area with larger pages to reduce translation overheads.

The straightforward implementation of external communication is expensive because the hypervisor intervenes when physical events occur (e.g., interrupts or device accesses). Each such intervention entails an expensive "exit" from the guest code to the hypervisor. The highest-performing approach for reducing this overhead is device assignment, which eliminates exits for device access. Thus, to reduce these costs, we assign the network device directly to the guest using an SR-IOV-enabled adapter [23] which allows the guest to send requests directly to the device. To eliminate exits for interrupts, we use polling instead of interrupts, a well-known technique in storage systems.

To reduce the cost of internal communication, we modified KVM's para-virtual block driver to poll as well, eliminating exits due to PIOs and interrupt injections. This provides for a fast, exit-less, zero-copy transport.

By using these techniques, we show no measurable difference in network latency between bare metal and virtualized I/O and under 5% difference in throughput. For internal communication, micro-benchmarks show 6.6μs latency overhead, read throughput of 357K IOPS, and write throughput of 284K IOPS; roughly seven times better than a base KVM implementation. In addition, an I/O intensive filer workload running in KVM incurs less than 0.4% runtime performance overhead compared to bare metal integration.

Our main contributions are:

- a detailed, benchmark-driven analysis of virtualization overheads in a storage system context,
- a set of approaches to removing overheads, and
- a demonstration of how these approaches enable running new storage features in a VM with essentially zero runtime performance overhead.

The rest of the paper is organized as follows. Section 2 provides background on KVM and VM I/O. We take an incremental approach to show our performance improvements; Section 3 describes the experimental environment. Sections 4 and 5 present a performance analysis and describe optimizations related to the external and internal communication interfaces, respectively. Base overheads are shown together with macro-benchmark results are in Section 6. Section 7 describes related work and we conclude in Section 8.

2 x86 I/O Virtualization Primer

We now provide some background information on KVM (the hypervisor used in this paper) and virtual machine I/O. There are two main options for where a hypervisor resides. Type 1 hypervisors run directly on the hardware, whereas type 2 hypervisors are hosted by an OS. KVM takes a hybrid approach that combines the benefits of both. It is a Linux kernel module that leverages Intel VT-x or AMD-V CPU features for running unmodified virtual machines, thereby creating a single host kernel/hypervisor that runs both processes and virtual machines. Such a hybrid architecture allows the storage controller software to run unmodified on bare metal while also running additional functionality in virtual machines.

There are three main methods for accessing I/O devices in VMs. In the first, *emulation*, the hypervisor emulates a specific device in software [35]. The OS running in the VM (guest OS) uses its regular device drivers to access the emulated device. This method requires no changes to the guest, but suffers from poor performance.

In the second method, *para-virtualization* [4], the guest OS runs specialized code to cooperate with the hypervisor to reduce overheads. For example, KVM's paravirtualized drivers use *virtio* [26], which presents a ring buffer transport (*vring*) and device configuration as a PCI device. Drivers such as network, block, and video are implemented using virtio. In general, the guest OS driver places pointers to buffers on the vring and initiates I/O via a Programmed I/O (PIO) command. The hypervisor directly accesses the buffers from the guest OS's memory (zero-copy). Para-virtualized devices perform better than emulated devices, but require installing hypervisor-specific drivers in the guest OS.

The third method, *device assignment* [6, 17, 39], gives the VM a physical device that it can submit I/Os to without the hypervisor's involvement. An I/O Memory Management Unit (IOMMU) provides address translation and memory protection [6, 7, 38]. Interrupts, however, are routed to the guest OS via the hypervisor. Assigning a device to the VM means that no other OS can access it (including the hypervisor or other guests). However, technologies such as Single Root I/O Virtualization (SR-IOV) [23] allow devices to be assigned to multiple OSs.

3 Experimental Setup

We take an incremental approach to showing how to eliminate the virtualization overheads. For our experiments we used two servers, each with two quad-core 2.93GHz EPT-enabled Intel Xeon 5500 processors, 16GB of RAM and an Emulex OneConnect 10Gb Ethernet adapter. The servers were connected with a 10Gb cable. One server acted as a load generator and the other was our (emulated) storage controller platform.

We used RHEL 5.4 with the RedHat 2.6.18-164.el5 kernel for both the load server and the guest. The controller server used the RedHat kernel for bare metal runs and Ubuntu 9.10 with a vanilla 2.6.33 kernel for KVM runs. The newer kernel was necessary for running KVM.

The controller server was run with four cores enabled, unless otherwise specified. For VM-based experiments, two cores and 2GB of memory were assigned to the guest; all four cores were used by the host in the bare metal cases.

We used an 8GB ramdisk for the storage back-end in the experiments described in Section 5 and 6. This allowed us to measure I/O performance without physical disks becoming the bottleneck. We accessed the ramdisk via a loopback device, which allowed us to assign disk I/O handling to specific cores, similar to the way a storage controller functions.

All results shown are the averages of at least 5 runs, with standard deviations below 5%.

4 Network Communication Performance

Enabling the guest to interact with the outside world requires I/O access. As discussed in Section 2, each of the three common approaches to I/O virtualization has benefits and drawbacks. We identified device assignment—the best performing option—as the most suitable approach for adding new functionality to storage controllers. KVM's initial device assignment implementation, however, did not provide the necessary performance. In the remainder of this section, we analyze device assignment and discuss a set of optimizations which allowed us to achieve near bare-metal performance.

Virtualization overhead is mainly due to events that are trapped by the hypervisor, causing costly *exits* [1, 5, 16]. The overhead is a factor of the frequency of exits and the time it takes the hypervisor to handle the exit and resume running the guest. To examine the performance impact of virtualization for our intended use and ways to reduce it, we focused on networking micro-benchmarks. Our goal is to minimize the amount of time that the hypervisor needs to run, by minimizing the number of exits.

The first technique that we used to improve the guest's performance is related to the handling of the `hlt` (halt) and `mwait` x86 instructions. When the OS does not have any work to do it can call these instructions to enter a

power saving mode. Most hypervisors will trap these commands and will run other tasks on the core. In our case, however, the new function should always run. We therefore instructed the guest OS to enter an idle loop when there is no work to be done by enabling a kernel boot parameter (`idle=poll`). This improves performance, as the guest is always running.

The second technique that we used is related to interrupt handling. Most of the guest exits related to device assignment are caused by interrupts [2, 5, 18]. Every external interrupt causes at least two guest exits: first, when the interrupt arrives (causing the hypervisor to gain control and to inject the interrupt to the guest) and when the guest signals completion of the interrupt handling (causing the host to gain control and to emulate the completion for the guest). The guest can configure the adapter to use two different interrupt-delivery modes: MSI, which is the newer message based interrupt protocol, or the legacy INTX protocol. The KVM implementation we used incurred additional overhead when using MSI interrupts, due to additional exits for masking and unmasking adapter interrupts. Since most of the virtualization overhead comes from interrupts, our approach is to run the adapter in polling rather than interrupt-driven mode.

In Linux today, most network adapters use NAPI [30, 31], a hybrid approach to reducing interrupt overhead which switches between polling and interrupt-driven operation depending on the network traffic. However, even with NAPI, we have seen interrupt rates of 70K interrupts per second. Since such a high interrupt rate can incur prohibitive overhead and interrupts are not necessary for our intended use case, we decided to forgo interrupts and use polling. Our polling driver creates a new thread for the polling functions. The adapter we use has three types of events: packet received, packet sent, and command completion. Since there is no way to know when a packet will be received, our polling driver continuously polls for packets received; packet sent and command completion indications are handled by the same polling thread every so often. Using a constantly polling thread means that we dedicate most of a core for this functionality. While this might seem expensive from the resources perspective, it proved critical to achieve the desired performance. A single core could also be used to poll multiple devices by integrating their polling threads into a single thread, or by scheduling different polling threads on the same core. We did not experiment with this configuration.

Next we evaluate the performance of the polling driver using network micro-benchmarks. Table 1 depicts the average duration time of a ping flood command going from a client machine to the system under test. The system under test replies to pings using our polling driver either in *polling* mode or in *INTX* mode. The driver runs either in the host (*bare-metal*), or in the *guest* with halt disabled, or in the guest with halt enabled (*guest halt*).

	Bare-metal	Guest	Guest halt
INTX	24	49	89
Polling	21	21	21

Table 1: Ping average latency (µs)

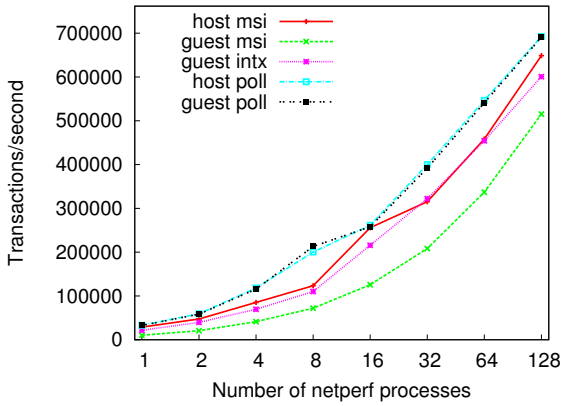


Figure 1: `netperf` request-response throughput

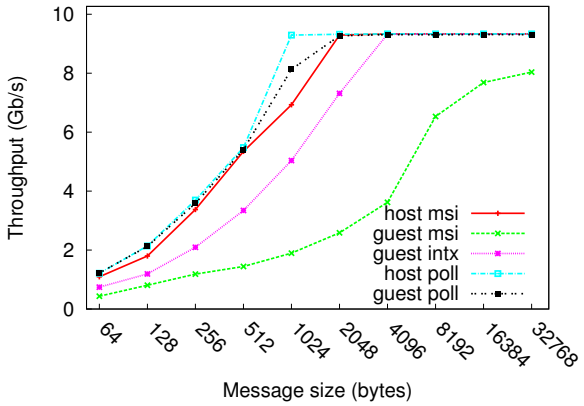


Figure 2: `netperf` TCP send throughput

Figure 1 shows the results for several `netperf` request-response configurations, measuring round-trip time using 1 byte packets. *guest msi* and *guest intx* stand for the guest using MSI and INTX interrupt delivery, respectively. *host msi* stands for the host using interrupts in MSI mode; *guest poll* and *host poll* stand for the guest and host using polling mode, respectively. As expected, polling mode achieves better performance than interrupt mode in the host (i.e, on bare metal). Since in guest mode the cost of interrupts is much higher, the gain from using polling is more significant than in the bare-metal case. Using MSI interrupts in guest mode has significant impact on the performance with this KVM version since there are frequent exits due to interrupt masking calls by the guest.

Figure 2 shows the results of a single-threaded `netperf` send TCP throughput test (system under test is sending) in the same configurations as the previous figure: host using polling and INTX interrupts, guest using

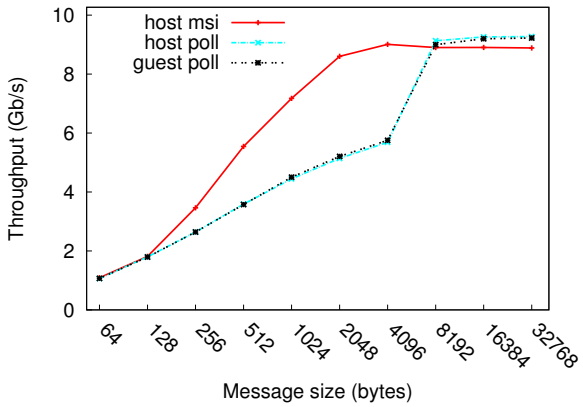


Figure 3: `netperf` TCP receive throughput

polling, INTX, and MSI interrupts. Here the contribution of polling is less noticeable, since the TCP stack batches network processing. On bare metal, polling provides better performance than interrupt mode. In guest mode, the advantage of polling is much more significant.

Figure 3 shows the results of a single-threaded `netperf` receive TCP throughput test (system under test is receiving). Here it is surprising to see that for the bare-metal case the performance of polling (*host poll*) is less than that of interrupts (*host msi*). The reason is that in the netperf throughput test the sender is the bottleneck. When the receiver is working in polling mode, it sends many more acknowledgment packets to the sender. For example in the case of 1K messages, the receiver sends approximately 10 times as many ACKs. Since the sender is already the bottleneck, sending more ACKs generates more load on the sender, which reduces sender throughput. While this issue is noticeable for this micro-benchmark, in practice, the handling time of a packet by the receiver is much larger, hence in most cases the sender is not the bottleneck. Polling achieves the same performance in the *guest poll* and *host poll* cases, which indicates that the virtualization-induced runtime overhead is negligible.

To verify that the reduced polling performance for the receive test is an artifact of TCP, we ran the same test using UDP. With UDP, all setups—guest or bare metal, interrupts or polling—achieve the same performance. Because the sender is the bottleneck, once the TCP ACK effect is removed, performance is not affected by the receiver's mode of operation.

5 Internal Communication Performance

Of the three methods for accessing I/O devices described in Section 2, we use para-virtualization for internal communication. Para-virtualization performs better than emulated devices, and because we supply the VM image that runs in the controller, we can easily use custom drivers. Further, our goal is to transmit I/O requests to a controller process running on the host, so device assignment is less

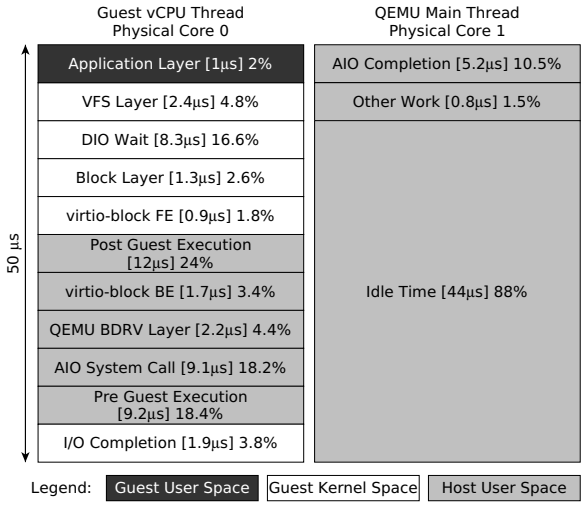


Figure 4: Unmodified KVM para-virtualized block I/O path.

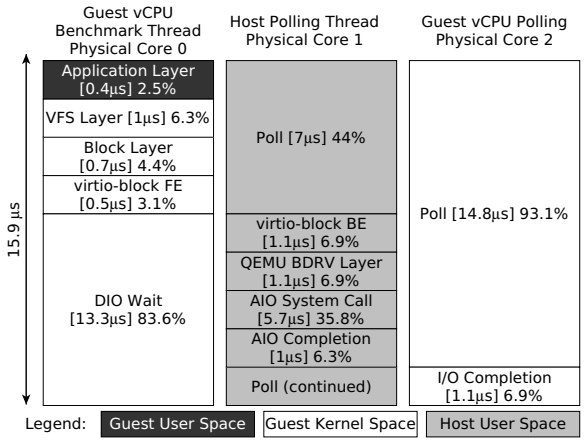


Figure 5: Para-virtualized block I/O path with polling.

practical. For example, we cannot assign the drives because the storage controller must "own" them, and not the guest OS. One may also consider using external communication to access the controller via iSCSI or Fibre Channel, but this adds unnecessary communication overheads.

We use ramdisk as the backing store for our analysis to prevent the disks from dominating latencies or becoming a bottleneck. In addition, we use direct I/O to prevent caching effects that mask virtualization overheads. Latencies presented are the average over 10 minutes.

Section 5.1 describes the vanilla KVM para-virtualized block I/O, and Section 5.2 describes our optimizations.

5.1 KVM Para-virtualized Block I/O

Figure 4 depicts the unmodified para-virtualized block I/O path in KVM, along with associated latencies for major code blocks when executing a 4KB direct I/O write request. The guest application initiates an I/O, which is handled by the guest kernel as usual. The direct I/O waiting time (DIO Wait, 16.6% of the total), consists of world

switches and context switches between threads inside the guest. Though we have drawn it as one block, it is interleaved with other code running on the same core. The para-virtualized block driver (virtio-block front-end) iterates over the requests in the elevator queue and places each request's I/O descriptors on the *vring*, a queue residing in guest memory that is accessible by the host. The driver then issues a programmed I/O (PIO) command which causes a world switch from guest to host.

Control is transferred to the KVM kernel module to handle the exit. The post- and pre-execution times (24% and 18.4%, respectively) account for the work done by both KVM and QEMU to change contexts between the guest and QEMU process (including the exit/entry). KVM identifies the cause of the exit and, in this case, passes control to the QEMU virtio-block back-end (BE). It extracts the I/O descriptors from the vring without copies and passes the requests to the block driver layer (QEMU BDRV), which initiates asynchronous I/Os to the block device. The guest may now resume execution.

An event-driven dedicated QEMU thread receives I/O completions and forwards them to the virtio-block BE. The BE updates the vring with completion information and calls upon KVM to inject an interrupt into the guest, for which KVM must initiate a world switch. When the guest resumes, its kernel handles the interrupt as normal, and then accesses its APIC to signal the end of interrupt, causing yet another exit. Locks to synchronize the two QEMU threads incur additional overhead.

5.2 Para-virtualized Block Optimizations

To reduce virtualization overhead, we added a polling thread to QEMU as depicted in Figure 5. The thread polls the vring (1) for I/O requests coming from the guest and (2) for I/O completions coming from the host kernel. The polling thread invokes the virtio-block BE code on incoming I/Os and completions. This thread does not necessarily need to reside in QEMU; if the storage controller is polling-based, its polling thread may be used.

As discussed in Section 4, we added a thread to the guest which polls the networking device. We utilize this same thread to poll the vring for I/O completions. When it detects an I/O completion event, it invokes the guest I/O completion stack, which would normally be called by the interrupt handler. By using polling on both sides of the vring, we avoid all I/O-related exits, and thus also avoid all of the pre- and post-guest execution code. We also avoid locking the queue, since now only the polling thread accesses it. For the 4KB direct I/O write, this improves the latency from 50μs to 15.9μs.

Comparing Figures 4 and 5, we see that polling better utilizes the CPU for I/O-related work. Additionally, components that we didn't directly optimize (such as the VFS layer, for example) are more efficient thanks to better cache utilization and less cache pollution due to fewer context switches.

We performed two additional code optimizations in QEMU to reduce latencies, whose impact is already included in the above discussion. When accessing a guest's memory, QEMU must first translate the address using a page-table–like data structure. This handles cases where the guest's memory can be remapped (for example, when dealing with PCI devices). In our case, the memory layout is static, rendering the translation unnecessary. Removing unnecessary lookups improved performance by 4.6% for 4KB reads and 4.2% for 4KB writes. The second optimization is to use a memory pool for internal QEMU request structures. This saved 3% for 4KB reads and 2.5% for 4KB writes.

5.3 Overall Performance Calculation

A storage controller running a new function in a VM that uses interrupts for its internal communication would have a rather significant performance penalty. Looking at Figure 4, the corresponding storage controller implementation would look similar, except that the AIO calls would be replaced by asynchronous calls to the controller code. We consider any work done from the time the application submits the I/O until it reaches the controller to be virtualization overhead (work that would not be done if running directly on the host). In the unmodified case, the overhead is 49μs (we subtract only the latency of the application layer from the total).

If our techniques were integrated into a controller, we would calculate the latency overhead as follows, based on Figure 5. We begin with the total, 15.9μs, and subtract the application layer, as we did in the previous case. Further, we subtract the QEMU BDRV layer, and the AIO system call and completion, because these would be replaced by the controller code, and are therefore not considered virtualization overhead. The final overhead is therefore 7.7μs before the two QEMU optimizations, and 6.6μs after.

To put the overheads in context, we estimate our performance impact on the fastest latencies published using the SPC-1 benchmark since 2009 [34]. The fastest result was 130μs, and our virtualization technique would add approximately 5% overhead to this case (the baseline case would add approximately 38%). The average of the 27 controllers' fastest latencies is 482μs, and in this average case, our virtualization techniques would add only 1.4% (the baseline would add over 10%).

Our improvements affect throughput in addition to latency. To measure these effects, we ran micro-benchmarks consisting of multi-threaded 4KB direct I/Os. For multi-threaded 4KB direct I/Os, we improved read IOPs by a factor of 7.3x (from 48.8K to 357.5K), and write IOPS by 6.5x (from 43.8K to 284.1K).

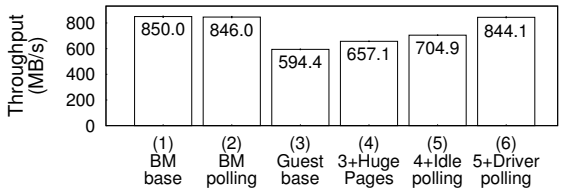


Figure 6: File server workload with 6 cores

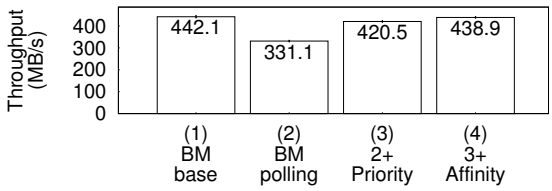
6 File Server Workload

We next tested the end-to-end performance of running a server in a VM on a storage controller. We ran a file server in our VM, and used dbench [36] v4.00 to generate 4KB NFS read requests which all arrived at the 10Gb NIC, went through the local file system and block layers, through the para-virtualized block interface, and were satisfied by the ramdisk on the host side. We always allocated two cores to the controller function, and either two or four to the file server (as specified). In the virtualized cases, all file server cores were given to the VM. All cores were fully utilized for all cases.

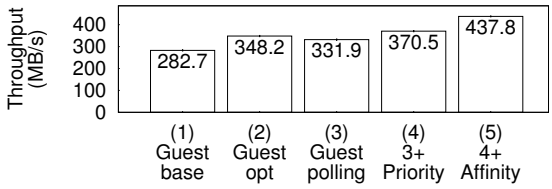
Figure 6 shows the results when running with six cores: two for the controller function and four for the file server. Bars 1 and 2 show the bare metal case without polling and with, respectively. Roughly the same performance is attained in both cases. The third bar shows the baseline measurement for the guest, which is a significant degradation as compared to the bare metal cases. We identified three main causes for this performance drop.

First, we noticed a large number of page faults on the host caused by the running VM. We mitigated this using the Linux kernel's *HugePages* mechanism, which backs a given process with 2MB pages instead of 4KB pages. This allows the OS to store fewer TLB page entries, resulting in fewer TLB faults and fewer EPT table lookups. HugePages improved performance by 10.5%, as shown in the fourth bar of Figure 6. A feature in a recent Linux kernel release makes the use of HugePages automatic [10]. The second issue affecting performance was halt exits, described in Section 4. We avoid these exits by setting the guest scheduler to poll. This further improved performance by 7.3% (fifth bar in Figure 6). The final performance improvement was to add driver polling, for both the network and block interfaces (described in Sections 4 and 5.2). This further improved performance by 19.7%, and brings the guest's performance to be statistically indistinguishable from bare metal.

Next, we ran the same workload, but this time allocated only two cores to the file server (four cores total). This may be a more common deployment when running multiple server VMs on a single physical host, for example, because there are less cores available for each VM. The bare metal results are depicted in Figure 7(a). The first bar shows the bare metal baseline performance of 442.1 MB/s. We see in the second bar that performance



(a) Bare Metal



(b) Guest

Figure 7: File server workload with 4 cores

drops to 331.1 MB/s when using polling. This is because the host now has only two cores, and the polling thread utilizes a disproportionate amount of CPU resources as compared to the file server. We remedied this by reducing the CPU scheduling priority of the polling thread (bar 3), and by setting the CPU affinities of the polling thread and some of the file server processes so that they share the same core (bar 4). These two changes bring the performance back to baseline performance.

In the guest case, depicted in Figure 7(b), the baseline (bar 1) is approximately 36% lower than the bare metal case. Bar 2 includes the HugePages and idle polling optimizations previously described, and bar 3 adds driver polling. Similar to the bare metal case, we adjusted the polling thread scheduling priority and the affinities of the relevant processes (bars 4 and 5). This brings us to results that are statistically indistinguishable from bare metal. In all cases, tuning was not difficult, and a wide range of values provided the achieved performance.

7 Related Work

Several works explored the idea of running VMs on storage controllers. The IBM DS8300 storage controller uses logical partitions (LPARs) to enable the creation of two fault-isolated and performance-isolated virtual storage systems on one physical controller [12]. Pivot3 [24] and ParaScale [22] are integrated virtualization and scale-out SAN storage platforms that are geared to data centers. Fido [8] investigated using shared memory to implement zero-copy inter-VM communication in Xen in the context of enterprise-class server appliances. Our focus is different in that we investigate external communication, zero-copy communication with the controller software, and various techniques and methods to reduce overheads caused by I/O virtualization.

Block Mason [21] used building blocks implemented in VMs to extend block storage functionality. VMware

VSA [37] pools the internal storage resources of several servers in a shared storage pool, using dedicated virtual machines running on each server.

Several works explored off-loading I/O to dedicated cores [3, 15, 16, 19]. The closest to ours is VPE [19], which adds host-side polling to KVM's virtio network stack. The VPE thread polls the network device for incoming packets and polls the guest device driver for new requests. However, the guest incurs exit overheads for interrupts and I/O completions since its driver does not poll. Dedicating cores for improving I/O performance has also been explored in TCP onloading [25, 32, 33].

There have been several works that investigated reducing interrupt overhead. The Linux kernel uses NAPI to disable interrupts of incoming packets as long as there are packets to be processed [30, 31]. A hybrid approach is to use interrupts under low load, and polling when more throughput is needed [11]. With interrupt coalescing, a single interrupt is generated for a given number of events or in a pre-defined time period [2, 27]. A series of works compared these techniques qualitatively and quantitatively [28, 29]. Rather than polling for fixed intervals or according to arrival rates, QAPolling uses the system state as determined by applications' receive queues [9]. The Polling Watchdog uses a hardware extension to trigger interrupts only when polling fails to handle a message in a timely manner [20].

ELI (ExitLess Interrupts [13]) is a recently-published software-only approach for handling interrupts within guest virtual machines *directly* and *securely*. ELI removes the host from the interrupt handling paths, thereby allowing guests to reach 97%–100% of bare-metal performance for I/O-intensive workloads.

8 Conclusions and Future Work

We have shown how to use a hypervisor to host and isolate new storage system functions with negligible runtime performance overhead. The techniques we demonstrated such as polling, dedicated cores, avoiding page lookups, etc., while not general purpose are a good fit to our usage scenario and have a significant payback.

There are several possible extensions. First, ELI [13] is a promising new approach for exitless interrupts which would remove the need to poll in the guest. We are investigating incorporating it into our system. Second, if we stay with polling, we can explore ways to better utilize the polling cores, e.g., to on-board the TCP stack to a polling core. Third, we can also benchmark these techniques when running multiple VMs. Finally, we can examine how to leverage the fact that we have virtualized the new storage function's implementation to take advantage of features such as VM migration to improve performance and availability.

Acknowledgments

Thank you to our shepherd Arkady Kanevsky and the reviewers for their helpful comments. Thanks to Zorik Machulsky and Michael Vasiliev for assisting with the hardware. The research leading to the results presented in this paper is partially supported by the European Communitys Seventh Framework Programme ([FP7/2001-2013]) under grant agreement number 248615 (IOLanes).

References

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2006.

[2] I. Ahmad, A. Gulati, and A. Mashtizadeh. vIC: Interrupt coalescing for virtual machine storage device IO. In *USENIX Annual Technical Conf.*, 2011.

[3] N. Amit, M. Ben-Yehuda, D. Tsafrir, and A. Schuster. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conf.*, 2011.

[4] 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 *ACM Symposium on Operating Systems Principles (SOSP)*. ACM SIGOPS, October 2003.

[5] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B. Yassour. The Turtles project: Design and implementation of nested virtualization. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, 2010.

[6] 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 *Ottawa Linux Symposium (OLS)*, July 2006.

[7] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. van Doorn. The price of safety: Evaluating IOMMU performance. In *the 2007 Ottawa Linux Symposium*, June 2007.

[8] A. Burtsev, K. Srinivasan, P. Radhakrishnan, L. N. Bairavasundaram, K. Voruganti, and G. R. Goodson. Fido: Fast inter-virtual-machine communication for enterprise appliances. In *USENIX Annual Technical Conf.*, June 2009.

[9] X. Chang, J. Muppala, W. Kong, P. Zou, X. Li, and Z. Zheng. A queue-based adaptive polling scheme to improve system performance in gigabit ethernet networks. In *IEEE International Performance Computing and Communications Conf.*, 2007.

[10] J. Corbet. Transparent hugepages in 2.6.38. `http://lwn.net/Articles/423584/`, January 2011.

[11] C. Dovrolis, B. Thayer, and P. Ramanathan. Hip: Hybrid interrupt-polling for the network interface. *ACM SIGOPS Operating Systems Review*, 35(4):50–60, October 2001.

[12] B. Dufrasne, A. Baer, P. Klee, and D. Paulin. IBM system storage DS8000: Architecture and implementation. Technical Report SG24-6786-07, IBM, October 2009.

[13] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, D. Tsafrir, and A. Schuster. ELI: Bare-metal performance for I/O virtualization. In *Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2012.

[14] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *the 2007 Ottawa Linux Symposium*, volume 1, June 2007.

[15] S. Kumar, H. Raj, K. Schwan, and I. Ganev. Re-architecting VMMs for multicore systems: The sidecore approach. In *Workshop on Interaction between Opearting Systems & Computer Architecture (WIOSCA)*, 2007.

[16] A. Landau, M. Ben-Yehuda, and A. Gordon. SplitX: Split guest/hypervisor execution on multi-core. In *USENIX Workshop on I/O Virtualization (WIOV)*, 2011.

[17] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, 2004.

[18] J. Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2010.

[19] J. Liu and B. Abali. Virtualization polling engine (VPE): Using dedicated cpu cores to accelerate I/O virtualization. In *the 23rd international conference on Supercomputing*, June 2009.

[20] O. Maquelin, G. R. Gao, H. H. J. Hum, K. B. Theobald, and X. Tian. Polling watchdog: Combining polling and interrupts for efficient message handling. *ACM SIGARCH Computer Architecture News*, 24(2):179–188, May 1996.

[21] D. T. Meyer, B. Cully, J. Wires, N. C. Hutchinson, and A. Warfield. Block mason. In *USENIX Workshop on I/O Virtualization (WIOV)*, 2008.

[22] ParaScale. Cloud computing and the parascale platform: An inside view to cloud storage adoption. White Paper, 2010.

[23] PCI-SIG. Single root I/O virtualization 1.1 specification, January 2010. `www.pcisig.com/specifications/iov/single_root/`.

[24] Pivot3. Pivot3 serverless computing technology overview. White Paper, April 2010.

[25] G. Regnier, S. Makineni, I. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. Tcp onloading for data center servers. *IEEE Computer*, 37(11):48–48, 2004.

[26] R. Russell. virtio: Towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, July 2008.

[27] K. Salah. To coalesce or not to coalesce. *International Journal of Electronics and Communications*, 61(4):215–225, 2007.

[28] K. Salah, K. El-Badawi, and F. Haidari. Performance analysis and comparison of interrupt-handling schemes in gigabit networks. *Journal of Computer Communications*, 30(17):3425–3441, 2007.

[29] K. Salah and A. Qahtan. Implementation and experimental performance evaluation of a hybrid interrupt-handling scheme. *Journal of Computer Communications*, 32(1):179–188, 2009.

[30] J. Salim. When NAPI Comes To Town. In *Linux 2005 Conf.*, August 2005.

[31] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond Softnet. In *Anual Linux Showcase & Conf.*, 2001.

[32] L. Shalev, V. Makhervaks, Z. Machulsky, G. Biran, J. Satran, M. Ben-Yehuda, and I. Shimony. Loosely coupled TCP acceleration architecture. In *The 14th IEEE Symposium on High-Performance Interconnects*, August 2006.

[33] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. IsoStack—highly efficient network processing on dedicated cores. In *USENIX Annual Technical Conf.*, June 2010.

[34] SPC. Storage Performance Council, 2007. `www.storageperformance.org`.

[35] J. Sugerman, G. Venkitachalam, and B. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conf.*. USENIX Association, 2001.

[36] A. Tridgell and R. Sahlberg. DBENCH. `http://dbench.samba.org/`, 2011.

[37] VMware. Virtual storage appliance. `http://www.vmware.com/products/datacenter-virtualization/vsphere/vsphere-storage-appliance/overview.html`.

[38] P. Willmann, S. Rixner, and A. L. Cox. Protection strategies for direct access to virtualized I/O devices. In *USENIX Annual Technical Conf.*. USENIX Association, June 2008.

[39] B. Yassour, M. Ben-Yehuda, and O. Wasserman. Direct device assignment for untrusted fully-virtualized virtual machines. Technical report, IBM Research Report H-0263, 2008.