FastXenBlk: high-performance virtualized disk IOs without compromising isolation

Damien Thenot, Jean-Pierre Lozi, Gaël Thomas

FastXenBlk: high-performance virtualized disk IOs without compromising isolation (industry track)

Damien Thenot
Vates SAS
Grenoble, France
damien.thenot@vates.fr

Jean-Pierre Lozi
Inria
Paris, France
jean-pierre.lozi@inria.fr

Gaël Thomas
Télécom SudParis - IP Paris
Palaiseau, France
gael.thomas@telecom-sudparis.eu

ABSTRACT

Optimizing IO in a type I hypervisor such as Xen is difficult because of the cost of exchanging data between a VM and the driver. We address this challenge by proposing FastXenBlk, a new IO driver for Xen. FastXenBlk uses three mechanisms to improve IO performance. First, it uses several threads that poll multiple virtual IO queues that are exposed to a guest in order to execute IOs in parallel. Second, it batches requests in order to minimize the number of hypercalls to Xen. And third, it uses kernel bypass in order to avoid system calls during IOs. We evaluate FastXenBlk using the FIO benchmark with different access patterns and IO sizes. Our evaluation shows that FastXenBlk consistently improves the latency and the throughput for all workloads as compared to tapdisk, the driver currently used in production, by a factor of up to 3×.

ACM Reference Format:
Damien Thenot, Jean-Pierre Lozi, and Gaël Thomas. 2023. FastXenBlk: high-performance virtualized disk IOs without compromising isolation (industry track). In *Proceedings of 24th ACM/IFIP International Middleware Conference (Middleware 2023 Industrial Track) (Middleware '23).* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

While a few years ago, the cost to access storage mainly came from the hardware, this is not the case anymore. Modern servers switched from SATA I solid state drives (SSDs) with a maximum bandwidth of 0.15 GB/s[1] to NVMe devices that are directly connected to the PCIe bus with a bandwidth of up to 3.5 GB/s. As a result, the software mechanisms used to virtualize the storage devices now play a critical role in terms of performance.

With a modern device, optimizing the IO path in a type I hypervisor is particularly difficult. A type I hypervisor, such as Xen [3], enforces security by running the device drivers inside an isolated virtual machine (VM) named dom0. By isolating the drivers, a type I hypervisor drastically reduces the attack the surface as compared to a type II hypervisor (e.g., KVM/Qemu), which runs the hypervisor on top of a full kernel [4]. For example, in Xen, the attack surface is limited to a small virtualization layer of only 337 thousand lines of C code, while the attack surface of KVM/Qemu includes Linux and its 24 million lines of C code (70 times more lines of code). However, due to the secure design of Xen, the memory of the VMs is not visible from dom0 [8]. The device driver located in dom0 thus cannot configure a device to directly access the memory of a VM that performs an IO. The driver can only read and write data from/to the memory of dom0, which means that any IO leads to a costly operation to exchange the read or written data between the memory of dom0 and the memory of the VM that performs the IO. With a modern device, this operation to exchange data significantly slows down IOs. For example, with tapdisk, the main IO driver of Xen, we measure that exchanging data for a sequential read of 1 MiB contributes to more than 80% of the execution time.

In this paper, we explore how we can optimize IO in a type I hypervisor without breaking the isolation between dom0 and the VMs. To that end, we propose to combine three different techniques. The first technique consists in increasing the throughput by running multiple IOs in parallel. To that end, we propose to expose multiple virtual IO queues in the VM, and to use multiple threads in dom0 to execute the IOs.

The second technique consists in batching the operations that are used to exchange data between a VM and dom0. We measure that for small IOs, the overhead of exchanging data mostly comes from the cost of performing a hypercall to a copy or remap function in Xen. By batching the hypercalls to this function, we significantly decrease the cost of exchanging data.

The last technique consists in optimizing the control path as much as we can. The driver can either run inside the kernel of dom0, or as a userland process in dom0. We choose the second option due to industrial considerations, since doing so eases (i) the deployment of the driver as it does not require changes to the Linux source code, and (ii) the implementation of advanced features such as snapshotting. We bypass the kernel of dom0 by using the Storage Performance Development Kit (SPDK) [9]. SPDK is a userland library which makes it possible to bypass the kernel of dom0 by directly mapping the hardware queues of a storage device inside a userland process—the driver, in our design. Thanks to SPDK, the driver executes each IO in dom0 without requiring system calls or a copy between the driver and the kernel of dom0.

We implement our three optimizations in a new driver for Xen named FastXenBlk. FastXenBlk consists of 1,600 lines of code. At a high level, FastXenBlk is a userland process in dom0 which (i) runs multiple threads to handle multiple virtual IO queues in parallel, (ii) relies on batching to minimize the number of hypercalls to Xen, and (iii) relies on SPDK to bypass the kernel of dom0. FastXenBlk

[1] Up to 0.6 GB/s with SATA III, the latest version of SATA.

Middleware '23, December 11–15,2023, Bologna, Italy

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

	Parallelism		Usability	Isolation	Performance	
Driver	**Virtual Queues**	**Threads**	**Features**	**Isolated from VM**	**Reception**	**Control path**
blkback	multiple	single	simple	yes	kernel	blkback → hardware
tapdisk	single	single	advanced	yes	userland	tapdisk → libaio → hardware
virtio-blk	multiple	single	advanced	no	userland	qemu → libaio → hardware
FastXenBlk	multiple	multiple	advanced	yes	userland	FastXenBlk → hardware

orange: kernel module blue: userland process

Table 1: Main virtual IO devices

is being developed as part of our effort to improve IO performance in the XCP-ng distribution of Xen.[2]

We evaluate FastXenBlk with FIO [2]. FIO covers a wide range of access patterns, which highlights how FastXenBlk will behave in production, with real workloads. Our evaluation shows that when FastXenBlk exposes 4 virtual queues to the guest VM, it has a an at least 2.88× higher throughput than tapdisk, the driver currently used in production in Xen. FastXenBlk can reach a throughput of 1,438 MiB/s, which represents 65% of the maximal possible throughput that we observe when running FIO in native Linux.

We also compare FastXenBlk with KVM/Qemu and its state-of-the-art virtio-blk device driver. As already stated, KVM/Qemu is a type II hypervisor with a much larger attack surface than Xen. However, the type II design is more efficient for IOs than the type I design since, in the type II design, the device driver has full access to VM memory. The driver can thus configure the storage device to directly access the memory of the VMs that use the device, which avoids the costly operation that is used in a type I hypervisor to exchange data between dom0 and the VM. Our results show that the throughput of FastXenBlk is between 3× lower and 1.7× higher than the throughput of virtio-blk. We identify that the lower performance of FastXenBlk is due to the type I hypervisor design, which trades direct access to VM memory for better isolation. This result shows that, with modern storage devices, there is a trade-off between the strong isolation of type I hypervisors and the high performance of type II hypervisors.

To summarize, this paper presents the following contributions:

- It proposes FastXenBlk, a virtual disk IO driver for Xen that uses parallelism, batching, and kernel bypass to optimize IO performance in a type I hypervisor.
- It evaluates FastXenBlk with FIO, and shows that, thanks to the three optimizations described above, FastXenBlk has at least a 2.88× higher throughput than tapdisk, the driver that is currently used in production by Xen.
- It shows that FastXenBlk remains up to 3× less efficient than virtio-blk due to the cost to exchanging data between dom0 and the VM that perform the IOs.
- It shows that the performance of modern storage devices introduces a new trade-off between security and performance.

The remainder of the paper is organized as follow. §2 presents the background, §3 the design of FastXenBlk, and §4 the evaluations, §5 related work, and §6 the conclusions.

2 BACKGROUND

The software architecture used to virtualize a storage device consists in three components: a communication channel, a frontend driver, and a backend driver. The communication channel connects the frontend driver to the backend driver. The frontend driver is located inside the guest operating system of a VM. It exposes an interface to access the storage to the guest operating system, and forwards the requests to the backend driver. The backend driver is located inside the host system for a type II hypervisor, and inside the dom0 VM for a type I hypervisor. It receives the requests from the frontend and execute them on the real hardware.

The remainder of this section discusses several implementations of this overall design that were proposed for KVM/Qemu and Xen.

2.1 Communication channels

This section presents the main communication channels used in Xen and in KVM/Qemu.

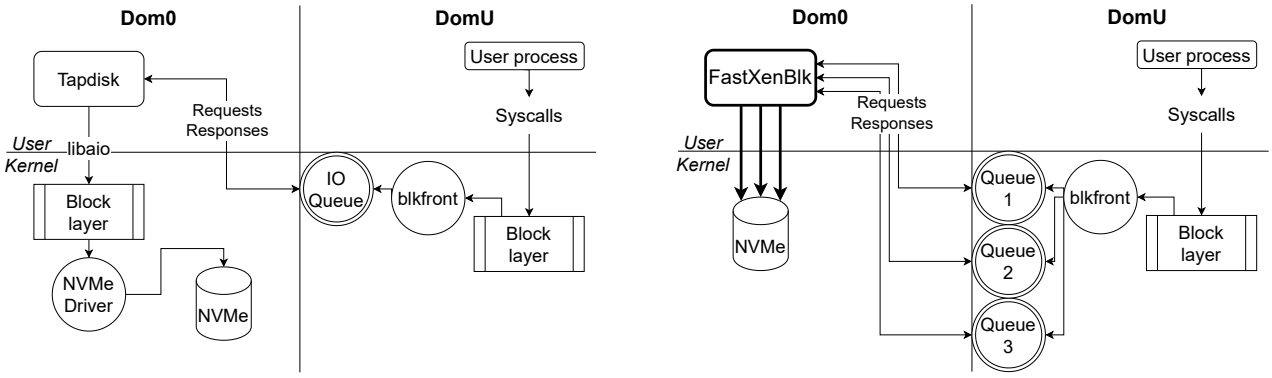
2.1.1 XenStore (Xen). In Xen, all VMs can access a key-value store named the *XenStore*. The XenStore is located in dom0. It interprets a key as a path, and a VM is only granted access to the values associated to a subset of the paths. To access data, a VM sends a get or put request to dom0, and triggers the execution of the request in dom0 by sending an IRQ.

2.1.2 Grant tables (Xen). Communicating through the XenStore is inefficient because of the IRQ, which triggers an hypercall to Xen and then an upcall to dom0. Accessing the XenStore is also inefficient because the XenStore handles the put and get requests sequentially. For this reason, the XenStore is often used during boot time to establish more efficient communication channels. To that end, Xen provides *grant tables*.

A grant table is a structure used by a granter (a VM or dom0) to expose regions of its physical address space. In detail, a grant table is an array of entries that is indexed by a grant identifier. An entry contains the identifier of a grantee (another VM or dom0), and a description of the shared memory region. Typically, at boot time, a granter inserts an entry into the grant table and sends the grant identifier to the grantee by registering it into the XenStore. When the grantee finds the grant identifier in the XenStore, it performs a hypercall to map or copy the memory region of the granter in its physical address space. In the case of a map, typically, the granter and the grantee do not use the XenStore anymore: they only communicate through a ring buffer that is stored inside the shared memory region.

2.1.3 Virtio (KVM/Qemu). In KVM/Qemu, communication is implemented by using a *virtio* device. A virtio device is a virtual PCIe device that is exposed in the physical address space of the guest. The guest sends requests by writing inside the memory of the virtio device, and receives responses by reading its memory. The requests

[2]https://xcp-ng.org/



(i) tapdisk (ii) FastXenBlk

Figure 1: Tapdisk and FastXenBlk architectures.

are transmitted through a ring buffer in a memory region that is shared between the host and the guest.

Since KVM/Qemu already has full access to the memory of the VMs, it does not need a mechanism to dynamically create new memory regions that are shared between the host and the guest. This property makes communication in KVM/Qemu simpler than in Xen. The ring buffers of virtio are sufficient to exchange messages between the host and the guest, and, if the message contains pointers to the memory of the guest, the host directly uses them to access the memory of the guest.

2.2 Frontend drivers

In our study, we focus on the most efficient frontend drivers for an NVMe device in Xen and Linux/KVM.

2.2.1 Blkfront (Xen). Blkfront is a Linux kernel module. It exposes a classical block device interface. It can handle one or multiple virtual IO queues. Each virtual queue is a memory region that is shared with the backend driver through the grant tables.

A virtual queue is implemented as a ring buffer. It is used to exchange requests between the VM and dom0. A request contains the type of operation (read, write or discard), a location on the storage device, and a set of grant identifiers, which allows dom0 to access the IO buffers of the VM. Because the size of a request is limited to 4 KiB, a request cannot contain more than 11 grant identifiers, which translates to request of at most 44 KiB.

When blkfront prepares a request, it adds the IO buffers one by one to the grant table. It removes them from the grant table when dom0 sends an acknowledgment.

2.2.2 Virtio-blk (KVM/Qemu). Virtio-blk [1] is a Linux kernel module that also exposes a classical block device interface. As blkfront, virtio-blk can handle one or multiple virtual IO queues, which allows the guest to execute IOs in parallel.

Instead of virtio-blk, a guest can use virtio-scsi, which considers that the virtual PCIe exposed by virtio implements the SCSI protocol. However, virtio-blk is more efficient than virtio-scsi for NVMe devices, which are the focus of our work. Therefore, we do not consider virtio-scsi in the rest of this paper.

2.3 Backend drivers

Several backend drivers have been proposed for KVM/Qemu and Xen. As shown in Table 1, they implement different trade-offs between parallelism (e.g., support for multiple virtual queues), usability (e.g., support for disk snapshots), isolation (between the driver and the VM), and performance (e.g., shorter control path).

2.3.1 Blkback (Xen). blkback is a backend for blkfront. It is implemented as a Linux kernel module for dom0 in Xen. It supports multiple virtual IO queues and exhibits excellent parallelism. Since blkback runs inside the kernel, its control path is especially simple: it accesses the storage from the kernel directly. In terms of features, blkback is, however, particularly lacking. It can only use one full physical partition on the storage device. It also cannot take a snapshot of the disk, and thus cannot be used for VM migration. Advanced features could be implemented inside blkback, but due to the large amount of code that would be required, the Xen developers consider that it is preferable to implement such features in userspace. As a result, while blkback is integrated inside the kernel tree, it is only rarely used in production because it remains minimalistic in terms of features.

2.3.2 Tapdisk (Xen). Tapdisk is the default storage backend that is shipped with the XenServer and XCP-ng distribution. It offers advanced features such as disk snapshot, virtual disks, or disks stored on logical volume. As shown in Figure 1.(i), tapdisk is implemented as a userland process. It receives blkif requests from blkfront directly in userland. When it executes a request, it copies the IO buffers from/to the VM by using the grant tables. In order to execute an IO request, tapdisk has to call the kernel of dom0. To that end, it uses libaio, an interface of the Linux kernel which handles the IOs asynchronously. In term of parallelism, tapdisk is relatively inefficient. It only implements a minimal blkif interface with a single virtual queue, and it only uses a single thread to handle requests.

2.3.3 Vhost-blk (KVM/Qemu). Vhost-blk is a backend for virtio-blk. It runs inside Qemu and has direct access to the memory of the VM. It offers advanced features, and, like tapdisk, uses libaio to transfer the requests to the Linux kernel. In terms of parallelism, performance and features, virtio-blk is by far the most efficient

backend, especially because it does not have to copy or map the IO buffers in its address space to handle a request. In terms of isolation, as discussed in the introduction, the design of KVM/Qemu is not secure because vulnerabilities in KVM, Qemu, or virtio-blk may be exploited to maliciously access the memory of the VM.

3 DESIGN AND IMPLEMENTATION

As shown in the previous section, none of the existing backends provide isolation, performance, parallelism, and usability at the same time. In order to provide these features, we propose to combine different techniques: (i) the implementation of FastXenBlk as a userland process in order to ease its deployment in production and in order to implement advanced features, (ii) the use of multiple virtual IO queues and multiple threads to handle the queues, (iii) the use of batching to decrease the number of calls to Xen to exchange data between the VM and dom0, (iv) kernel bypass to avoid costly system calls, and (v) the implementation of FastXenBlk in dom0 by using the grant tables to enforce isolation.

Figure 1.(ii) presents the overall architecture. Like with tapdisk, FastXenBlk is a userland process that receives blkif requests directly from blkfront. Contrarily to tapdisk, which only handles a single virtual IO queue, FastXenBlk handles multiple IO queues with multiple threads. Moreover, while tapdisk relies on libaio to send the requests to Linux, which slows down the control path, FastXenBlk relies on SPDK. SPDK is a userland library which bypasses Linux by exposing the hardware IO queues in a process.

3.1 Polling threads and batching

SPDK uses its own lightweight scheduler inside the application. FastXenBlk integrates itself as an SPDK application by defining lightweight SPDK threads. FastXenBlk associates a lightweight polling thread to each virtual IO queue. A polling thread executes a request from the beginning to the end. FastXenBlk also associates each polling thread to a single hardware queue of the NVMe, which is used to actually execute the request.

In order to reduce the number of hypercalls used to copy the IO buffers, FastXenBlk batches calls to Xen. In detail, when a polling thread receives a request, it fetches all the requests of the virtual queue, and calls Xen once to copy or map the IO buffers.

A polling thread handles the requests asynchronously. It executes a loop, which (i) polls the pending requests from the virtual IO queues, (ii) handles the pending requests if they exist, (iii) polls the hardware completion queues of the NVMe, and (iv) acknowledges the completed IOs to the VM when requests are completed.

3.2 Grant table in FastXenBlk

In order to access the IO buffers of a VM, FastXenBlk can optionally copy or map the IO buffers.

3.2.1 Grant copy. When FastXenBlk is configured to copy the IO buffers, it handles a write request by first calling Xen in order to copy the IO buffers in large pages of 2 MiB inside its address space. FastXenBlk then forwards the request to the NVMe device through a hardware queue.

For a read request, FastXenBlk transforms the request in order to read data from the NVMe device into large buffers of 2 MiB that are located inside its address space. When FastXenBlk receives the completion message from the NVMe device, it copies its local memory to the IO buffers of the VM with a single hypercall.

3.2.2 Grant map. When FastXenBlk is configured to map the IO buffers, it maps them inside its address space just after having received a read or write request. Then, FastXenBlk forwards the request to the NVMe device, which accesses the memory of the VM in-place, without requiring an extra copy since the IO buffers of the VM are mapped inside dom0.[3] Moreover, upon reception of the completion message from the NVMe, FastXenBlk unmaps the IO buffers from dom0 by executing an hypercall.

3.3 Virtual versus physical addresses

FastXenBlk runs as a process in userland and thus uses guest virtual addresses. However, NVMe devices address the memory of dom0 by using guest physical addresses, not by using guest virtual addresses. Thus, FastXenBlk has to convert the virtual addresses of the buffers into guest physical addresses before emitting a request to the NVMe. To that end, FastXenBlk uses a mechanism provided by SPDK which relies on the `/proc/self/pagemap` interface provided by the Linux kernel. This interface is slow: each access to `/proc/self/pagemap` triggers a system call to inspect the page table of the process. For this reason, SPDK uses two techniques to minimize the number of accesses to `/proc/self/pagemap`. First, SPDK only considers large pages of 2 MiB. By only considering large pages, a single access to `/proc/self/pagemap` gives the physical addresses of 4,096 pages of 4 KiB. And second, SPDK maintains, in userland, a map which caches the physical addresses of 2 MiB-long IO buffers that are recycled between the IOs.

When FastXenBlk *copies* the IO buffers of the VM inside the memory of dom0, it can recycle the SPDK buffers, which allows SPDK to avoid costly accesses to `/proc/self/pagemap`. Unfortunately, when FastXenBlk *maps* the IO buffers of the VM inside the memory of dom0, it creates new buffers for which the physical addresses are not yet known. This leads to accesses to `/proc/self/pagemap`. Moreover, while SPDK has support for 2 MiB pages, a blkif request only contains pages of 4 KiB. For this reason, FastXenBlk has to use `/proc/self/pagemap` for each 4 KiB-page of a blkif request, which still slows down FastXenBlk when it maps the IO buffers.

4 EVALUATION

This section presents our experiments.

4.1 Hardware and software settings

We use a 2-socket machine Intel Xeon E5-2637 v2 CPUs @ 3.50GHz and an Intel Optane SSD 900P NVMe drive. For the Xen experiments, we use the XCP-ng Xen distribution version 8.2. It includes Xen version 4.13.4 (XCP-ng release 9.21.2.xcpng8.2) for the virtualization layer, and a custom XCP-ng CentOS-based distribution of Linux for dom0 (Linux 4.19.19 release 7.0.15.3.xcpng8.2, gcc 4.8.5, glibc 2.17 release 222.el7, and tapdisk 3.37.3 release 1.0.1.xcpng8.2). The VM runs an Arch Linux distribution (Linux 5.18.15, gcc 12.1.1, and glibc 2.36). For the Linux/KVM experiments, the guest runs the same Arch Linux distribution, and the host runs Debian 11 (Linux

[3]The NVMe directly accesses the physical address space of dom0 because the IOMMU maps the guest physical address of dom0 to the host physical addresses of the machine.

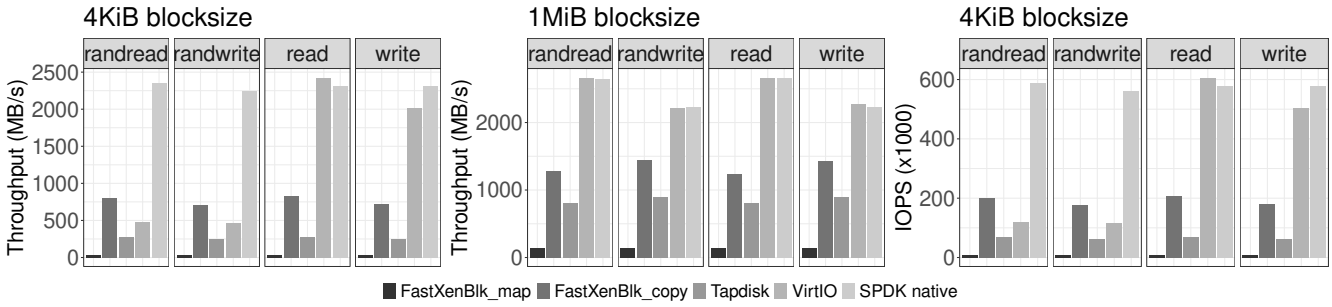


Figure 2: Throughput of the backends.

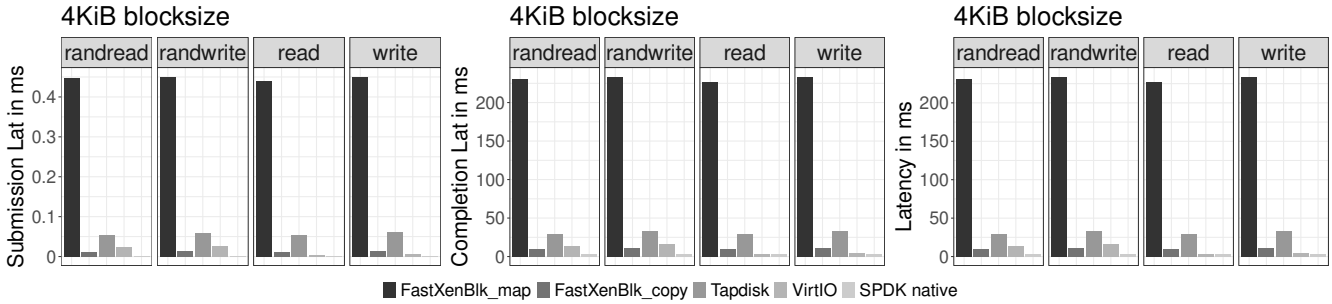


Figure 3: Latencies of the backends for 4 KiB accesses.

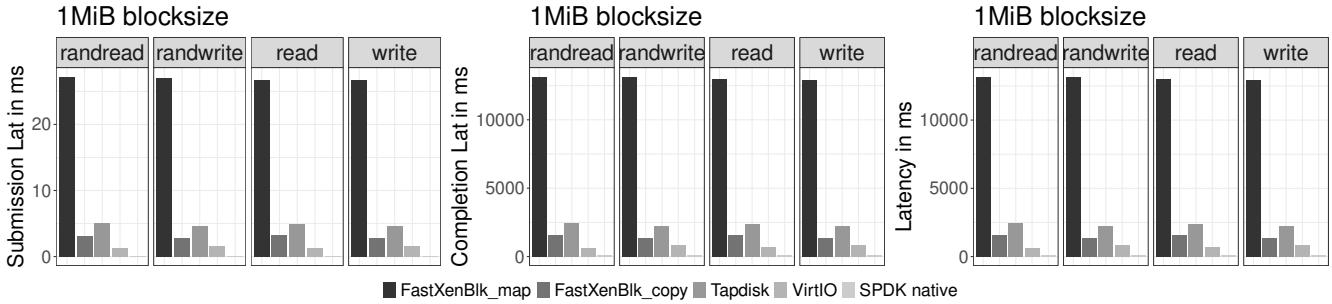


Figure 4: Latencies of the backends for 1 MiB accesses

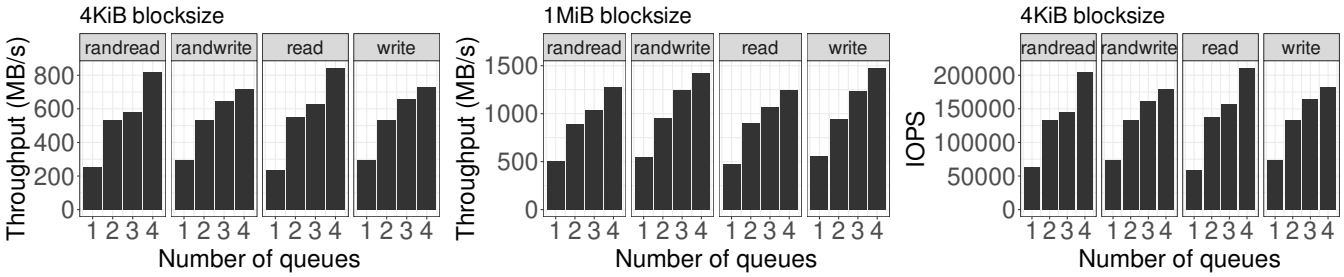


Figure 5: Scalability of FastXenBlk_copy with multiple queues.

5.10.179, and glibc 2.35) with Qemu 5.2.0. In all experiments, the drive is formatted as ext4.

We evaluate FastXenBlk in two flavors: FastXenBlk_copy copies the data from the VM to dom0 (see §3.2.1) while FastXenBlk_map

dynamically remaps the IO buffers (see §3.2.2). We compare FastXenBlk with tapdisk in Xen, with virtio-blk in KVM/Qemu, and with a native version that maximizes the throughput by using SPDK. In Figures 2, 3 and 4, FastXenBlk_copy, FastXenBlk_map and virtio-blk expose 4 virtual IO queues to the guest VM. In Figure 5, we vary the number of virtual IO queues.

We evaluate the different configuration with the state-of-the-art disk benchmarking tool FIO version 3.29. FIO runs 4 processes. Each process accesses 5 GiB of data by using a single thread. Therefore, in total, a run accesses 20 GiB of data. We present the performance with IOs of 4 KiB and of 1 MiB block sizes in order to study how the drivers react with both small and large IOs. Moreover, we present experiments with both sequential and random access patterns, as they exhibit different performance behaviors.

4.2 Comparison between the backends

FastXenBlk_copy versus tapdisk. In Figure 2, we observe that with small IOs of 4 KiB, FastXenBlk_copy exhibits a speedup in terms of throughput of 2.88× to 3.02×, as compared to tapdisk, thanks to batching and better parallelism. In Figure 3, we observe that FastXenBlk_copy also divides latency by 2.94× to 3.13× thanks to kernel bypass (i.e., the use of SPDK). For large IOs of 1 MiB, FastXenBlk_copy exhibits a speedup in terms of throughput of 1.51× to 1.61× (Figure 2). FastXenBlk also divides latency by 1.52× to 1.64× (Figure 4) in this case. Overall, these results show the benefits of our different optimizations.

FastXenBlk_copy versus FastXenBlk_map. In Figures 2, 3 and 4, we also observe that FastXenBlk_copy significantly outperforms FastXenBlk_map. With FastXenBlk_map, FastXenBlk has to map the read and written memory regions, and then it has to unmap them, which multiplies by two the number of hypercalls to Xen as compared to FastXenBlk_copy (see §3.2.1 and §3.2.2). Moreover, as discussed in §3.3, in FastXenBlk_map, SPDK has to retrieve the physical address of the pages after the remapping. This operation significantly slows down IOs as compared to FastXenBlk_copy, which uses pre-mapped SPDK buffers of 2 MiB.

FastXenBlk_copy versus virtio-blk. In Figures 2 and 4, we observe that virtio-blk is almost at the level of native with large buffers of 1 MiB. With a large buffer, the time to forward the request from the VM to the storage device becomes negligible as compared to the time spent in the storage device to copy the data between the device and the memory. Since the NVMe device directly accesses the memory of the VM with virtio-blk, the performance is almost as good as that of native. We also observe that, with a large buffer, the throughput of FastXenBlk_copy is between 1.5× to 2× lower than with virtio-blk. By using Linux perf, we identify that almost all the overhead is caused by the hypercall that is used to copy memory between dom0 and the VM. This result highlights the cost of exchanging data between a VM and dom0 in a type I hypervisor.

In Figures 2 and 3, we observe that FastXenBlk_copy outperforms virtio-blk in terms of throughput and in terms of latency for random reads and random writes. This is caused by the optimized IO path of FastXenBlk_copy: thanks to kernel bypass, FastXenBlk_copy avoids costly communication between FastXenBlk and the kernel, while virtio-blk relies on the libaio interface of the host kernel to execute IOs. As shown by Yang et al. [10], replacing libaio in KVM/Qemu eliminates this cost.

With sequential reads and writes, virtio-blk still performs 1.5× to 2.9× better than FastXenBlk_copy in terms of throughput and latency. With a sequential access pattern, the guest kernel quickly detects that prefetching data is useful. It thus sends large IO requests, and thus behaves similarly as with large IOs of 1 MiB.

4.3 Scalability

Figure 5 presents the scalability of FastXenBlk_copy. We observe that adding more virtual IO queues in FastXenBlk_copy increases the throughput, regardless of the configuration (random or sequential, read or write, or small or large IOs). This result is explained by the use of threads to handle the IO queues in FastXenBlk, which execute the submitted IOs in parallel.

5 RELATED WORK

One of the notable works on Xen is the implementation of virtio over grants, which is an ongoing effort in the Xen community [5]. This implementation is at its early stage, and therefore, we can not yet compare the performance of FastXenBlk and virtio over grants. The virtio over grants implementation relies on ARM features such as the use of MMIO transport to create the communication ring. As a result, this implementation is not easily portable to other architectures such as x86_64. This is not the case of FastXenBlk since FastXenBlk only relies on generic and portable components. Moreover, the current implemented backend of virtio over grants does not use kernel bypass, and since the code is not yet available, we do not know if it implements multiple threads or batching.

On KVM, other works lower the cost of multiplexing hardware. For example, Peng et al. [6] directly expose the NVMe hardware queues inside the guest, and require minimum request rewriting in order to share the devices among VMs [6]. However, implementing advanced features such as backup or migration seems difficult with this architecture because the VMs do not coordinate when they access the disk. With FastXenBlk, we use a centralized driver, which relies on SPDK and already provides such advanced features.

Using SPDK as a backend driver in KVM/Qemu was proposed by Yang et al. [10]. Their implementation has excellent performance, but it requires full access to VM memory. In FastXenBlk, we also use SPDK, but we preserve the isolation of a type I hypervisor.

Ram et al. [7] modified the grant table mechanism in order to reuse a shared buffer between IOs. They do not use multiple threads, batching or kernel bypass like FastXenBlk. Their solution also requires heavy modification of the VM's kernel, which makes it hard to deploy in production. We are currently investigating a similar solution to reuse shared buffers, but without requiring heavy modifications to the VM's kernel.

6 CONCLUSION AND FUTURE WORK

This paper presents FastXenBlk, a new disk IO driver for Xen. FastXenBlk uses a combination of parallelism, batching and kernel bypass to significantly increases performance as compared to tapdisk. FastXenBlk, however, remains slower than virtio because of the cost of isolating dom0 in Xen.

REFERENCES

[1] 2022. Virtual I/O Device (VIRTIO) Version 1.2. https://docs.oasis-open.org/virtio/virtio/v1.2/csd01/virtio-v1.2-csd01.html

[2] Jens Axboe. 2023. Fio-flexible I/O tester synthetic benchmark. https://github.com/axboe/fio. Accessed: 2023-07-10.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*.

[4] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. 2011. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP '11*. https://doi.org/10.1145/2043556.2043575

[5] Jürgen Groß. 2021. Virtio with Xen N.0 (N > 2). https://static.sched.com/hosted_files/xen2021/bf/Thursday_2021-Xen-Summit-virtio.pdf

[6] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. 2018. MDev-NVMe: a NVMe storage virtualization solution with mediated pass-through. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*.

[7] Kaushik Kumar Ram, Jose Renato Santos, and Yoshio Turner. 2010. Redesigning Xen's memory sharing mechanism for safe and efficient I/O virtualization. In *Proceedings of the 2nd Workshop on I/O Virtualization, Pittsburgh, PA, USA*. https://www.hpl.hp.com/techreports/2010/HPL-2010-39.pdf

[8] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. 2017. Deconstructing Xen.. In *2017 Network and Distributed System Security Symposium - NDSS '17*.

[9] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. https://doi.org/10.1109/CloudCom.2017.14

[10] Ziye Yang, Changpeng Liu, Yanbo Zhou, Xiaodong Liu, and Gang Cao. 2018. SPDK Vhost-NVMe: Accelerating I/Os in Virtual Machines on NVMe SSDs via User Space Vhost Target. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*. https://doi.org/10.1109/SC2.2018.00016