CCF Transactions on High Performance Computing (2020) 2:3–15
https://doi.org/10.1007/s42514-020-00023-y

REVIEW PAPER

Freezing time emulating new and faster devices with virtual machines

Luis C. E. Bona[1] · Alessandro Elias[1] · Andre P. Ziviani[1] · Ramon Nou[2] · Toni Cortes[3] · Marco A. Z. Alves[1]

Received: 14 October 2019 / Accepted: 6 February 2020 / Published online: 26 February 2020
© China Computer Federation (CCF) 2020

Abstract
Recent proposals of emerging data storage devices make it necessary to reevaluate all levels of the storage hierarchy to optimize the software stack performance. However, these new devices are not always widely available and therefore early experiments may be impossible. Emulators aim at mimicking as close as possible the behavior of a component, nonetheless, emulating new and fast storage devices is a challenging task due to time perception. In this work, we propose an approach to emulate storage devices using virtual machines (VMs) allowing the evaluation of a new device within a real system. We use a technique called freezing time, which pauses a VM to manipulate its clock and hide the real I/O completion time. Our approach is implemented at the hypervisor level and it is transparent to the guest operating system or application. We evaluate the technique under a real system using regular magnetic disks to emulate faster storage devices. Our method presented a latency error of 6.5% compared to a real device. Moreover, decoupled experiment between two laboratories, at the Barcelona Super Computing Center (BSC) in Spain, and the Center of Computer Science and Free Software (C3SL) in Brazil, demonstrated that our approach is reproducible and promising to allow the virtual evaluation of next-gen storage devices.

Keywords Virtual machine · Storage device · Dataplane thread · Emulation · Time dilation

1 Introduction

The storage gap motivates scientists and companies to investigates new ways to improve data store and retrieve performance. Part of the performance improvements for storage systems can be achieved at the software level, e.g., read-ahead operation using an intelligent storage adapter (Shah 2016), adaptive intelligent storage controllers and associated methods (Flower and Gajjar 2016). From a hardware point of view, for decades the main storage technology was based on magnetic disks and the performance gains were not disruptive.

These technologies demand a reevaluation of all layers of the storage hierarchy, raising a series of "what-if" questions. It is necessary to ensure that those technologies are worth it when they are integrated into all the input/output (I/O) stack, including workloads and real applications. Simulators can be used to circumvent the physical unavailability of the devices to be evaluated. However, such simulators are too slow to execute a detailed simulation of all the computer components running a full application.

Although simulation techniques are appropriate for some scenarios, it can hardly capture the interactions between different components, for example, the processor interactions with the operating system. Moreover, this technique does not allow to extrapolate the results to the real environment.

An alternative is to use emulators which aim at mimicking as close as possible the behavior of a component such that the emulated device is indistinguishable from the real one. However, emulating new and fast storage devices is a challenging task due to time perception issues since

✉ Alessandro Elias
aelias@inf.ufpr.br

Luis C. E. Bona
bona@inf.ufpr.br

Andre P. Ziviani
andrepziviani@gmail.com

Ramon Nou
ramon.nou@bsc.es

Toni Cortes
toni@ac.upc.edu

Marco A. Z. Alves
mazalves@inf.ufpr.br

[1] Universidade Federal do Paraná (UFPR), Curitiba, Brazil

[2] Barcelona Supercomputing Center (BSC), Barcelona, Spain

[3] Universitat Politecnica de Catalunya (UPC), Barcelona, Spain

the devices used as a back end for emulation are slower than the devices to be emulated. A storage emulator also will affect the latency and throughput observed as it takes time to process the requested I/O. One approach for such an emulator is to use the main memory as a backend (Lee et al. 2012). However, its capacity is still at least an order of magnitude smaller than that of storage devices. Another technique that can be used is the time dilation which consists in making the observed system clock move in slower steps which makes the emulated device relatively faster (Gupta et al. 2005), i.e., the time spent inside the guest is not the same as outside of it. If we apply a dilation of a 10×, the guest would see 1 second for every 10 s of “real-time”, and this will distort all the components.

Our approach for such storage device emulation is based on virtual machine (VM) technologies to run a full stack of software in the target emulated devices, so we do not need to face much of the performance penalties of a full-system simulator nor change the workloads nor the applications. To tackle time perceptions issue we employ a mechanism called freezing time (Bona et al. 2018) which explores the ability of a virtual machine to be paused and then resume execution. In this way, during the execution of the I/O requests a storage simulator or emulator can be called, meanwhile, time can be frozen without distorting any other components and can be done transparently for both the guest operating system and applications.

In this article we present the following main contributions, and extend our previous published paper (Bona et al. 2018):

- Flexible emulator tool for storage simulation which requires no changes in user space applications or changes in the kernel.
- Whole stack approach enabling analysis from the storage backend device up to the application running inside the guest.
- High precision emulation with an average latency difference of less than 7% considering read and write operations.
- Low overhead tool with less than 25% increase in emulation time, enabling fast evaluation of new and future storage devices.
- Open source software that is available at Elias (2019), under GNU General Public License (GPL).
- Simulation of a new device, Intel Optane DC Persistent Memory (DCPMM) extracting the latency results from another machine. We show that it is feasible to emulate them. More results and workloads are left for future work.
- A solution able to simulate a storage device that is faster than the memory of the host machine.

To make this experiment reproducible, evaluations in this article were executed in x86 architecture machinery, using open-source software. The evaluation shows we were able to emulate disks with RAM like speeds with an overhead of less than 25% in I/O request time while keeping precision as high as 94% on average.

2 Related work

Lee et al. (2012) is presented an emulation technique based on using a RAM disk as the storage backend. One of the disadvantages is that memory is small and expensive, so it can only execute small workloads. In our work, we can use any backend to simulate any other storage device.

Gu and Zhao (2012) is employed a technique that freezes the system disabling interruptions and the hardware clocks on the host kernel. However, another machine is needed to process the I/Os and, in order to communicate to this machine, the kernel’s network driver was modified to use a busy-wait system instead of an interrupt based system. The option to implement in the host’s kernel made it extremely dependent on the hardware used to implement.

The network time dilation approach was presented in Gupta et al. (2005), but using this method, all the components have their time distorted: effectively making every operation affected by a slowdown. This approach has an important drawback: The time dilation factor is proportional to the speed of the emulated device. This implementation also makes modifications to the virtual machine monitor (Xen_Project 2015) and the guest’s kernel.

Bayati et al. (2019) aims to evaluate Non-Volatile Memory Express (NVMe) devices in big data processing environments such as Apache Spark. To allows for testing NVMe related code without the need for buying expensive hardware they also propose a Quality of Service (QoS) aware NVMe emulator. This emulator has similarities with ours but they do not focus on improving the emulator accuracy as we do but rather on accurate modeling of the NVMe specific mechanisms.

Our emulator’s accuracy can be improved by reducing overhead, Kim et al. (2016) improves the performance of asynchronous I/O (AIO) in the VM. On that paper, the author used some of the same techniques that were used in this article: Virtual Machine Monitor (Quick Emulator, QEMU), VirtIO and dataplane access mode but they used a custom implementation of it which improved performance by 47% when compared to the standard dataplane mode. This technique could be applied to the freezing time approach but it will not be explored in this article.

3 Background

The implementation of our emulator is based on the combination of KVM (Kernel-based Virtual Machine) and QEMU (Quick EMUlator), which is an active project for more than 15 years, presenting low overhead and being widely adopted. KVM is a kernel module that gives access to hardware virtualization capabilities that allows processor virtualization with almost no overhead. Emulation of I/O devices is provided by QEMU and the VirtIO, a platform for IO virtualization, one of the de facto standards for this task. Another important aspect to understand the solution proposed here is time virtualization. In the remainder of this section, we present details on these aspects.

3.1 Virtual interface for I/Os

Emulating a real hardware device is costly because every single step needs to be processed as the real device would, which causes overhead in the host machine and reduces the throughput and Input/Output Per Second (IOPS) (Rizzo et al. 2013). *VirtIO* was created in a way that the guest and host could communicate directly or without having to emulate a real device that needs to stop the execution (*kick*) of the guest, therefore lowering the overhead and increasing the performance (Gordon et al. 2012). *VirtIO* presents itself to the guest as a PCI device, this way the guest only needs to implement a new PCI driver, and the Virtual Machine Manager (VMM) needs only to add virtual ring support to the devices they implement (Rizzo et al. 2013).

VirtIO unifies how virtual devices probe and configuration occur in the Linux Kernel allowing multiple implementations to be developed by different hypervisors. One of the components of this architecture is a common Application Binary Interface (ABI) for buffer publication and use. The *VirtIO_ring* implementation was deliberately conservative to avoid points that could be considered undesired by developers (Russell 2008).

The *VirtIO* driver is implemented with a separate abstraction level for drivers, transport, and configuration. Those abstractions are provided by a set of common helpers for the virtual driver which should be simple and as close as to optimal as possible providing efficient operation.

When a *VirtIO* device is found the *probe* function from the driver is called. The virtual device configuration happens in four steps: (1) reading and writing feature bits; (2) reading and writing the configuration space; (3) reading and writing the status bits; (4) device reset. The feature bits represent the features supported by a given virtual device, for example, the VIRTIO_NET_F_CSUM bit indicates whether a network device support checksum offload. When configuring the device the feature bits corresponding to the desired features must be set. Those bits are explicitly acknowledged by the guest, hence the host is sure about which features the driver understands. The next step is associated with the configuration space which is a structure containing device-specific information associated with a given virtual device that can be both read and written by the guest. The only requirement to add new features is to set the feature bit numbers and configuration space layout. Next, the guest indicates the status of the device probe through the status word (8 bits). On success, VIRTIO_CONFIG_S_DRIVER_OK is set showing that the feature probing phase was completed. As the final step, the device configuration and status bits are cleared (Russell 2008).

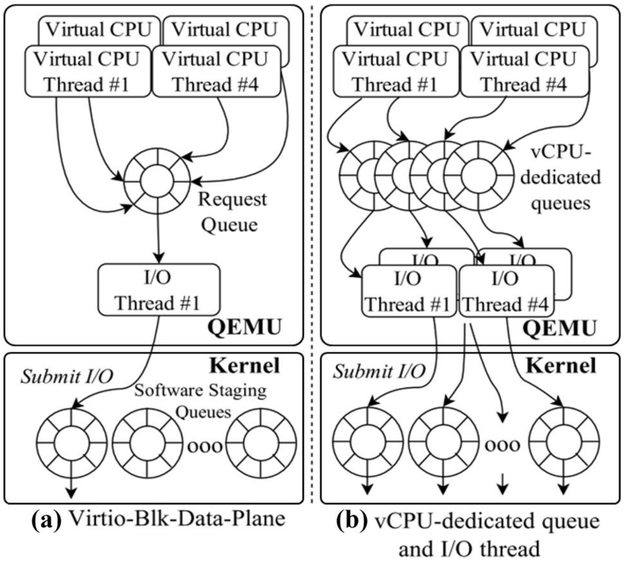


Fig. 1 Differences in the VirtIO-queue, investigating from vCPU until the device block (Kim et al. 2015)

The API *find_vq* populates the *virtqueue* structures attributing an index number for the *VirtIO* device. The *virtqueue* can be seen simply as a queue in which the guest posts buffers to be consumed by the host. Given the cost to notify the host is expensive, multiple buffers can be added simultaneously.

Each *VirtIO_ring* consists of three components: (1) the descriptor array; (2) the available ring; (3) the used ring. The descriptor is used by the guest to chain pairs of guest-physical address and length that are ready to use, the available ring indicates which descriptors chains are ready to be used and the used ring is like the available ring but is used by the host to indicate which descriptors were consumed.

Figure 1 represents the path the I/O takes in the system from the vCPU to the device. *VirtIO* block is a part of the *VirtIO* system and it is responsible for integrating those parts described above and exporting a block-like interface to the kernel. Other modules do the same thing for other peripherals like *VirtIO-net*, *VirtIO-gpu*, etc.

3.2 Timekeeping: choosing a clock source

The task of keeping a virtualized clock is nontrivial, especially over the x86 platform. One approach that offers great compatibility is to emulate an existing clock device. A disadvantage of this approach is the performance penalty since the device needs to be emulated, yet almost all current VMMs offer this option for compatibility reasons. Common options are HPET (High Precision Event Timer) (Amsden 2010a) and TSC (Time Stamp Counter) (Amsden 2010b). TSC mainly counts instruction cycles issued by the processor and is a good clock source since it has independent and dedicated circuitry and it is not affected by CPU clock changes. Sharing the TSC with the guest however is difficult since the guest would see time pass faster because the clock would still be running even when the VMM's process is not running on the host, making precise timing and interruptions inaccurate.

Live migration is also challenging for accurate timekeeping. First, along with the migration, the guest disables interruptions and meanwhile, time may need to be caught up. Later, the time may need to be adjusted considering it may now running at different rates. The destination host may have a faster TSC and it cannot be exposed to the guest due to the potential for time running faster than normal.

To solve those problems, *kvmclock* was designed. The approach is to register a memory page that store *kvmclock* data, so the VMM will write to it until explicitly disabled, or the guest is turned off. As the TSC is not emulated, nor the host's real clock source, the VMM writes multipliers and offsets compared to the host's TSC. In this manner the guest can adjust those values back into nanosecond resolution, with a small overhead, noticing only the time it was running. Those functionalities are wrapped in the functions KVM_GET_CLOCK and KVM_SET_CLOCK, and they are used on live VM migration.

When the VMM receives an I/O request, it will *kick* the guest and save its clock. Once the I/O request is ready, the VMM restores the clock and resumes the guest, so it believes that the time elapsed on this access was the latency of the I/O.

4 Freezing time storage emulator

The purpose of our freezing time technique is to enable the creation of a fast, efficient and transparent storage emulator, such as hard disk drives (HDD) or solid state disks (SDDs), that only dilates the time of the emulated device without interfering with other devices. This emulator allows complete comparisons through benchmarks without modifying or even recompiling the application. For the implementation, the KVM hypervisor was chosen because it has open-source and included in the Linux Kernel; *VirtIO* is another component of the solution as it is the I/O virtualization interface that is the current standard.

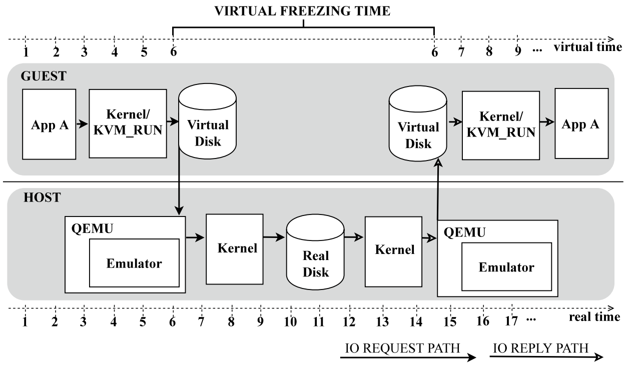


Fig. 2 Analyzing the path of an I/O request, from the guest application through the block device in the host (the time unit are meaningless, it is just a reference when the guest is "frozen")

Our main approach is to modify the host code (the VMM, QEMU) which will detect the occurrence of I/O requests as soon as possible and stop the guest execution. At this time with the frozen clock, the I/O requested is expected to be completed on the device to subsequently inject the emulated time and then allow the guest virtual CPU (vCPU) to be executed again (we will call this event as *KVM_RUN*).

The *KVM_RUN* is defined by KVM'S API among other calls such as *KVM_CREATE_VM*, *KVM_CREATE_VCPU*. The *KVM_RUN* call is used by the host to run a vCPU. The vCPU thread which issued the call will get blocked until an event interrupts the *KVM_RUN* call. This event is known as *kicking* the vCPUs from guest mode and has the consequence of returning back the control of vCPUs to QEMU, which allows us to manipulate the guest clock since it is not running.

Figure 2 represents the implementation proposed in this paper. The figure bottom half represents the host system and the real-time as perceived by the host system, the top half represents the virtual system (guest) and the virtual time perceived by it. The units of time measurement in this figure are meaningless and are only intended to show the clock behavior during the flow of an I/O request.

Initially, at real/virtual time 1, a user-space application generates an I/O request that will be received by the guest kernel, processed, and transformed into a request to be sent to the virtual disk driver at real/virtual time 6. Next, *QEMU* detects this request and *kicks* the guest. Note that at this time the guest clock will be frozen. Meanwhile, the request is processed and sent to the real disk. When the request is completed at real-time 14 (still virtual time 6), *QEMU* is notified via the host kernel and terminates the request handling and issues a *KVM_RUN*. This way the guest is resumed without perceiving the time taken to process the I/O. From this point on the request goes through the other steps until it reaches the application. Note that it is possible to delay notification of I/O request completion by allowing the emulation of devices with different performance parameters.

With our approach of pausing the time in the guest as soon as the I/O is detected and resuming the vCPUs as soon as it goes into context, we make the guest believes that the time has not elapsed, as shown in the gap from host time number seven until 14 on Fig. 2. Notice that we can eventually inject virtual time after the frozen period in order to simulate some specific device.

In the following section, we will show the implementation of the emulator based on *QEMU* version 2.5.0.

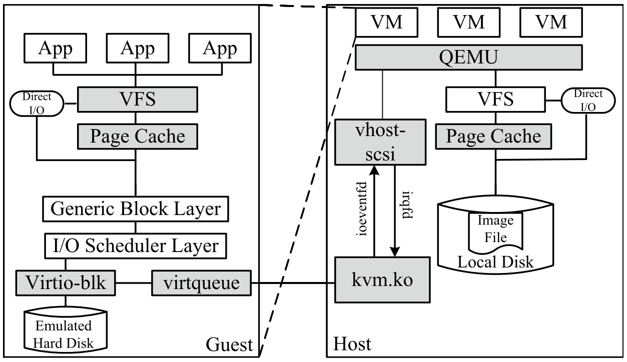


Fig. 3 Analyzing all layers of an I/O request. Adapted from Lu et al. (2016)

4.1 Implementation

We used *QEMU* version 2.5.0 to implement our emulator, which was the latest version at the time of the experiments. The only limitation to implement the emulator in other VMMs systems is the availability of source code. One of the first challenges was to understand how *QEMU* interacts with KVM: We have one thread per vCPU, one *IOThread* and other helper threads (that are not relevant in this context). Every time the VMM needs to issue a privileged operation (for example, access to the storage), it needs to *kick* the guest and get a global mutex to serialize every I/O operation. Once this operation finishes, the mutex is released, and the thread restarts the guest with a *KVM_RUN* (Gordon et al. 2012).

This workflow generates an important impact on performance, for this reason, since version 1.4, *QEMU* has a feature called *dataplane*. Thanks to this feature, a device has its own *IOThread*, so it did not affect the global mutex. Moreover, each device load is distributed in different CPU cores, if they are available, producing a reduced latency on each I/O operation. Both modes, *IOThread* and *dataplane*, are supported, but the latter is preferred due to its lower latency.

The *dataplane* thread gets blocked waiting for events in the FDs (File Descriptors) which represent its device. We added code right after this wait to identify if the event which causes the thread to unblock was an I/O request *popped* from the *VirtIO_ring*. In this case, the global mutex lock will be held avoiding concurrency with the *IOThread* or the vCPUs threads, next the guest will be kicked from execution and its virtual clock saved. At this point, the guest is out of context and the time elapsed from now on will be undetected by the guest. Then, a flush is issued causing the host to serve the requested I/O from the real system.

After finishing the I/O request the vCPUs threads are released and will wait at a barrier just before *KVM_RUN*. The global mutex lock is released unblocking other *QEMU* threads. The *dataplane* thread stays on a busy-wait waiting for vCPUs threads to reach the barrier. Then, each vCPU thread sets its clock back to the time that was registered when they were kicked and returns to *KVM_RUN* mode.

4.2 Analyzing the virtual I/O path, from the guest application to the host storage

Each guest I/O request goes through several layers from dispatch to arrival at the storage backend, Fig. 3 shows a simplified version of the flow of these requests across the I/O stages.

Whenever a process running on the guest generates an I/O request, the guest kernel checks to see if that request is already in the page cache. In the event of a page cache hit, the requested data is returned. Otherwise, the guest's kernel sends the request to the generic block layer through the I/O scheduler until it reaches the block device driver. The *VirtIO* block driver will dispatch the request from the virtual device to the host via *virtqueue*. On the host, the KVM kernel module detects the request and generates a notification via *vhost-scsi* to *QEMU*. The data plane is awakened and begins processing the request as if it were a regular process performing I/O on the local disk. When the host kernel finishes processing the I/O request, the QEMU process is notified and then it dispatches back the I/O response through *VirtIO*. Finally, the host notifies the guest *VirtIO* driver through the PCI bus, this way the request reaches the application that originated the request.

4.3 Emulating a storage device faster than the host memory

Emulating a faster storage device, Intel Optane DC Persistent Memory Module (DCPMM) from a remote machine, faster than the local memory speed could be cumbersome, the VM needs to go back in time (clock of the guest must be set to a time before the I/O happened). However, the following scenarios may occur when emulating it:

(i) The latency of the device being evaluated can be faster than the local memory speed, therefore the clock ticks must be adjusted back to when the I/O has been issued.
(ii) The latency of a specific virtual I/O operation in a faster storage device can be greater than the host storage device.
(iii) An error can occur. We will ignore this case since it is treated by the kernel and is out of context of our analysis.

Now, let's see an example of a DCPMM emulation using an HDD as a storage backend. Case (i), let's suppose an I/O A issues an interruption signaling that it is ready, and its latency is α. Since the samples extracted from the second machine are faster than α, let's say $\alpha - 1$ for a particular I/O (unit is meaningless). The I/O B we wish to emulate to behave just like A, such as with a latency of $\alpha - 1$, the clock needs to go back in time by a factor of one unit. Indeed, if we wish to adjust the latency, we must subtract the difference of latencies and make the clock back in time to represent the I/Os that occurred in the device DCPMM. As we can not predict exactly the differences in each I/O, the algorithm in the following paragraph shows the effectiveness with an accuracy of 90% on average.

Case (ii), the emulator could just use the sleep mechanism present in Sect. 4, after all the latency is ahead of the clock when the I/O was issued.

To achieve the emulation of a faster storage device, we used the mechanism of live migration offered by *QEMU*. This mechanism freezes all vCPUs when the live migration occurs. Once the VM is about to start in the target machine, the clock needs to catch up, *QEMU* stores the offsets of the clock and does the proper adjustments in the target machine. Before the VM starts and after adjustments of the *QEMU*, we have a chance to adjust the clock back in time.

To adjust the clock in case (ii) sleep can be used but for case (i), we have more limitations. Before we start emulating the latencies of DCPMM extracted from the second machine, we executed a set of experiments in the machine with DCPMM following the same methodology as described in Sect. 5, from those experiments we extracted the device behavior.

The algorithm is straightforward, we randomly generate a number between the minimum and maximum latency and after *QEMU* has adjusted the clock, we "scrum" it a little bit more, adjusting the clock according to the emulated latency of the DCPMM. However, this clock adjustment occurs only when an I/O is requested from the device that is being emulated in order to avoid freezing the guest when there are I/O requests from other devices. Also, the clock is adjusted only when the clock needs to go back in time, minimizing the overall effect in the VM.

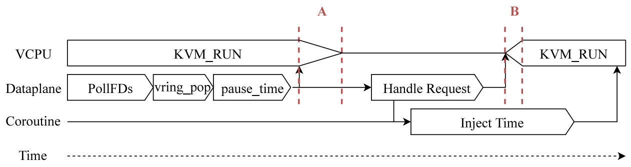


Fig. 4 Mechanism of the time dilation inside the VMM (*QEMU*)

4.4 Overhead of our emulator

The latency of our emulator is compound of two parts: the biggest is from the guest's kernel when it raises the request to the block device driver until the *dataplane's* thread process it and *kick* the guest. The small one is when the I/O is done, in the guest the clock is restored and in the host, VMM issues a *KVM_RUN*. We expected the time perceived by the guest to be zero, however it is not exactly zero. To get closer to zero, injection is made when the guest is frozen, so interruptions are not affected as time dilation occurs after all vCPUs are out of context.

Once a block device is set up to run in *dataplane* mode, fewer checks are needed, since the VMM allocated a thread with a local FD (File Descriptor) which is dedicated to poll I/Os for this specific block device, in this manner it is implicit which block device is been treated. One interruption is raised on the device when the guest pushes a request to the *VirtIO_ring*, so the host's kernel translates to an event on the FD. Once the *dataplane* thread pops the request from the *VirtIO_ring*, our emulator requests a global mutex lock to avoid the *IOThread* or vCPUs threads from trying to do a privileged operation and introduce entropy, checks if it is a data (read or write) request and, if it is, save the guest clock and *kick* it. In Fig. 4, the interval labeled A is the latency of *kicking* all the vCPUs, this procedure is not serial so the time from *kicking* the first and the last vCPU is greater than zero. Our emulator overhead is directly affected by this time distortion, this fact diminishes the achievable performance.

The VMM (*QEMU*) mitigates the problem with multiple call-back functions (Hajnoczi 2014) using the technique called *coroutine* (Knuth 1997). Our approach to adapt the coroutine is to sleep for a user-defined amount of time at the end of an I/O, so the device appears to have that latency to the guest. The timer is resumed after we issue *KVM_RUN* command.

The interval labeled B on Fig. 4, is the second part of the latency, at this point before issuing the *KVM_RUN* command we synchronize the vCPUs and restore the guest clock. B represents the latency of all vCPUs returning to execution.

In the following section we will validate the procedure described above through tracing tools and synthetic I/Os.

5 Experimental results

In this section, we show the evaluation of the emulator with a set of experiments. First, we showed the fastest I/O that technically could be achieved, using a RAM disk device Sect. 5.1. Then, in Sect. 5.2, our experiment shows the result of emulating the SSD using a slower storage backend, with modifications only in the VMM (*QEMU*). Finally, we determine the time elapsed in the experiment from the perspective of the guest and host (Sect. 5.5).

The results are presented simulating an SSD using the following devices: HDD (Hard Disk Drive) (model JPT39C), with a size of 1 TB, using SATA interface, speed of 3.0 Gb/s, 200 RPMs manufactured by Hitachi Global and Server Grade SSD Cloud Speed 500 (model TG32C1) manufactured by Smart Storage Systems. The selection was done to cover a mechanical device (HDD) and a non-mechanical device (SSD).

The host's and guest's operation systems were Debian Jessie, kernel version 4.4.4 installed on a separate HDD to not influence the experiments. The test bench machine was an AMD FX-6300 Six-Core Processor, 3.5 GHz, with 12 GB of DDR3. All the experiments executed in this section were performed on this specific system.

Synthetic workload took place using the tool *fio* (flexible I/O tester), which simulates a specific workload. It is configured by the user, such as sequential or random read/write, block size, number of threads, and so on. *Blktrace* (Block I/O layer tracing), is a block I/O layer tracing utility that provides the ability to collect detailed traces from the kernel for each I/O processed by the block I/O layer (Axboe 2007). *Blkparse* (Block I/O layer parser), parses the output events stored in files generated by *blktrace* in a human-readable way.

The VMM (*QEMU*) works with two types of backend support, *dataplane* (Oh et al. 2014) and *IOThread*. The first presented itself one order of magnitude more efficient than the *IOThread* mode, so the first has been chosen to execute all of the experiments.

The approach we followed in our experiments are:

1. Execute *blktrace* to collect I/O events. We are interested only in the response time.
2. While the above is executing, run the workload using *fio*, generating I/Os to the device we wish to evaluate.
3. When the workload finishes, stop the *blktrace* utility (thus saving all traces over the entire workload).
4. The pertinent I/O information is extracted from the traces saved by *blktrace* using the *blkparse* utility.

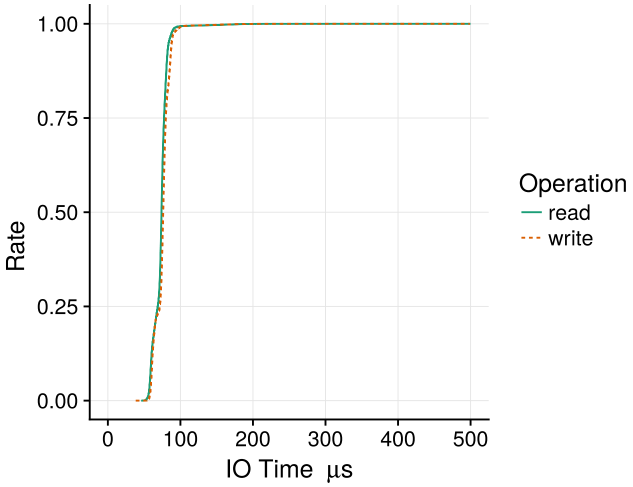


Fig. 5 Technically fastest I/O that can be reach on disk in RAM as backend; cumulative distribution chart

The order of the experiments is: running the *fio* program five instances, one at a time; on the RAM disk, SDD and HDD devices described above, operations, synchronously reading/writing. The data size of 4 GB with chunks of 4 KB to the backend storage device. The evaluation was executed with one and four vCPUs with a fixed amount of 2 GB of RAM in the guest. The cache was flushed after each iteration of the experiment, also *QEMU* was configured to not cache I/Os, to guarantee the effectiveness of each I/O issued through all the layers.

5.1 Empirical evaluation of the maximum device speed

In this first experiment, we seek to evaluate the limitations of the implementation of the time freeze technique. For this we will try to emulate I/O requests that are instantaneous, that is, with a time of completion equal to zero.

We setup *QEMU* to use the RAM as the storage backend to show the fastest I/O that can be achieved, as observed in Fig. 5 in CDF (Cumulative Distribution Function) format. The samples saturated at less than 100 µs. This value is the minimum I/O that can be achieved using RAM (fastest device) as a storage backend, values lower than that will need extra support as we introduced in Sect. 4.3.

5.2 Evaluation of the SSD

This section presents how the HDD and SSD react to the evaluation without the emulator, then emulate the SSD but using the HDD as the storage backend. A set of experiments was made with the SSD to empirically obtain the time to be injected, so this value was used to inject time at each I/O when simulating the SSD.

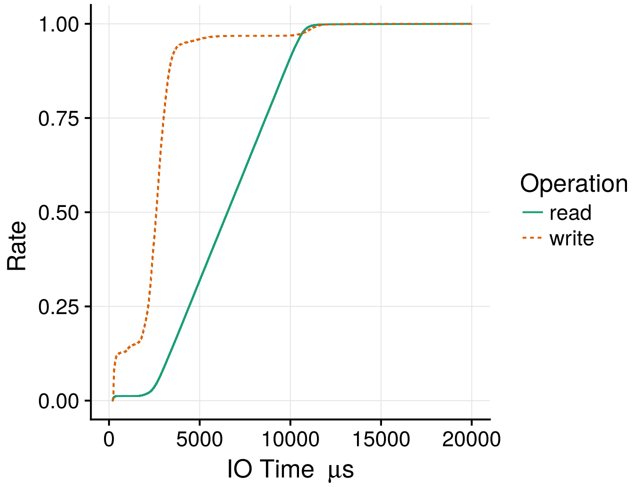


Fig. 6 Using HDD as backend with emulator off; 100% of write requests are below 5 ms and reads are between 2.5 and 10 ms

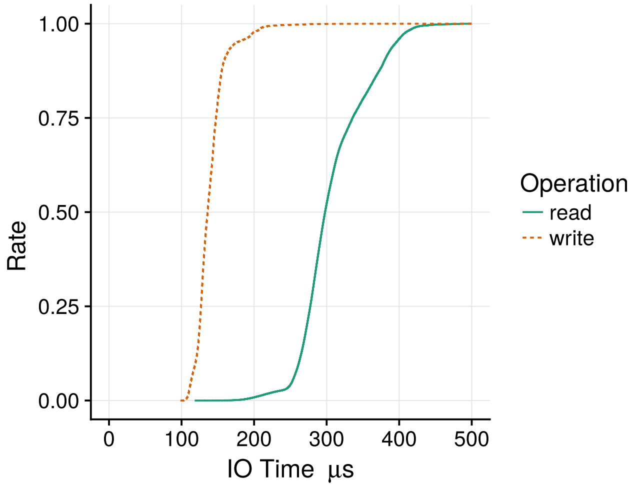


Fig. 7 Using SSD as backend with emulator off; 100% of write requests are below 200 and reads are between 200 µs and 400 µs

The Cumulative Distribution Function (CDF) in Fig. 6 shows the HDD behavior in the guest without any modifications in the VMM (*QEMU*). The I/O completion on the chart shows writes requests below 0.125 (or a 12.5%) are very fast due to the HDD buffer (before 2000 µs). Also it shows 100% of write requests are below 5 ms and reads are between 2.5 and 10 ms. The samples in the experiment show that HDD is multiple orders of magnitude more heterogeneous than the RAM or an SSD.

Figure 7 show the SSD behavior without the emulator, 100% of write requests are below 200 µs and reads are between 200 and 400 µs. It is closer to RAM devices than HDD but still near an order of magnitude worse.

In order to show the results of the experiments using our emulator to simulate the SSD using the HDD storage backend. Figure 8 aims our technique to provide results inside the observed variability of the original device. The results are divided by operation and by the number of vCPUs used and each box-plot[1] shows the distribution of the error difference between the mean SSD request and each of the I/O requests of the emulated SSD with the HDD backend (for each scenario). The original variability of the I/O requests in the SSD is shown with a transparent shade rectangle, as we can see the emulation is inside the rectangle on most of the scenarios.

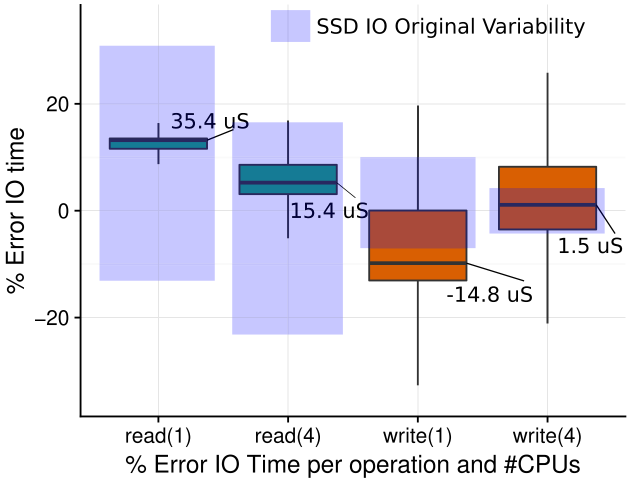


Fig. 8 The original variability of the I/O requests in the SSD is shown with a translucent rectangle, the emulation is inside the rectangle on most of the scenarios, even with a simple delay measurement. SSD emulation using an HDD as the storage backend

In these experiments, we show that it is possible to emulate an SDD using an HDD as a back end. The results obtained are quite satisfactory and show that the proposed implementation is feasible. The results could be better if the model to determine the emulated time of each I/O request was more complex since we used a very simplistic approach, which was the average service time.

5.3 Emulation of an intel DCPMM

Simulations of new or theoretical devices could be cumbersome to emulate. In the first case, device manufactures offers detailed information about the overall workload, the second case the model needs to determine such details. In our case study, we are investigating new technology, Intel Optane DC Persistent Memory Module (DCPMM). To achieve the

[1] Boxplots give an impression of how values of a group are distributed. The middle 50% of all values are within the box itself and the so-called whiskers have a length of at most 1.5-times the interquartile range.

Table 1 Minimum, maximum, average and standard deviation and percentiles (µs) of completed I/Os latencies while applying the time dilation mechanism, therefore freezing for the average value (2192.31 ns) of the latency of DCPMM (naive solution)

min = 132, max = 4365, average = 318.30, stddev = 47.08			
Percentiles			
1.00th = 171	5.00th = 205	10.00th = 274	20.00th = 298
30.00th = 310	40.00th = 318	50.00th = 326	60.00th = 330
70.00th = 338	80.00th = 346	90.00th = 366	95.00th = 378
99.00th = 418	99.50th = 430	99.90th = 478	99.95th = 506
99.99th = 588			

Table 2 Minimum, maximum, average and standard deviation and percentiles (ns) of completed I/Os latencies of the DCPMM from the Optane machine, obtained at BSC

min = 1614, max = 170885, average = 2192.31, stddev = 458.18			
Percentiles			
1.00th = 1768	5.00th = 1800	10.00th = 1848	20.00th = 1912
30.00th = 1944	40.00th = 1992	50.00th = 2096	60.00th = 2256
70.00th = 2352	80.00th = 2448	90.00th = 2544	95.00th = 2736
99.00th = 3120	99.50th = 3472	99.90th = 7776	99.95th = 8256
99.99th = 10048			

Table 3 Minimum, maximum, average, standard deviation and percentiles (ns) of completed I/Os latencies when emulating DCPMM on the Opteron machine at C3SL (storage backend in RAM)

min = 1642, max = 6709, average = 2577.73, stddev = 451.88			
Percentiles			
1.00th = 1784	5.00th = 1880	10.00th = 1960	20.00th = 2128
30.00th = 2256	40.00th = 2416	50.00th = 2576	60.00th = 2736
70.00th = 2896	80.00th = 3056	90.00th = 3184	95.00th = 3280
99.00th = 3376	99.50th = 3408	99.90th = 3440	99.95th = 3472
99.99th = 3600			

emulation precisely, cooperation between the two groups took place. At Barcelona Supercomputing Center (BSC) in Spain, they extracted the latency information through the *fio* workload, from the DCPMM storage device, present at the NEXTGenIO prototype. At Center of Computer Science and Free Software (C3SL) in Brazil, it was emulated the DCPMM storage device through the *freezing time* emulator, based on the information out of the benchmark from Barcelona.

Emulating a storage device like DCPMM is challenging, emulating very low latencies is difficult as in practice there are inaccuracies between stopping the clock and restarting it. To work around this problem we use an additional adjustment in the virtual clock as explained in Sect. 4.3.

For this experiment we employed the same methodology explained in Sect. 5. However, here we used a machine AMD Opteron Processor 6136 2.4 GHz, 32 cores compound of 8 NUMA nodes, 128 GB of RAM, memory manufacturer Samsung, DDR3 1333 MHz. The virtual storage device to emulate the DCPMM was a RAM disk of 16 GB. The VM was executed with 4 vCPUs, pinned in one of the NUMA nodes, limited to 2 GB of RAM. The *fio* workload is 10 GB of random read operations.

The observation about the problem of emulating a faster storage device than the one available in the target machine, raised when we ran with the naive solution. The mechanism of the time dilation is not accurate enough to emulate a device that is faster than a storage device in RAM. As observed in Tables 1 and 2, the average latency of the emulation in the Opteron machine is three orders of magnitude slower than DCPMM in the Optane machine at BSC. All the percentile shows a great difference in latencies in all ten slices, also the dispersion of the samples is observed by the standard deviation with a great difference.

Table 2 presents the latencies observed on the DCPMM extracted from the Optane machine at BSC. Those values are all target to emulate at C3SL on the Opteron machine.

Since the time dilation mechanism is not accurate enough, we used the new algorithm presented in the Sect. 4.3. In Table 2 we extract the parameters for the new proposed algorithm. The minimum and the maximum latencies are used to generate a random number between them, convert to a clock and adjust before the guest is started, therefore moving the clock back in time, mimicking the DCPMM.

In the following Table 3 we present the latencies when emulating the DCPMM on the Opteron machine at C3SL, with the storage backend in RAM.

The minimum latency presented on Table 3 is 98% similar of the min latency presented on Table 2. On the other hand, the max value is 2547% different. Nevertheless, if we analyze the statistics presented in Table 3 and compare with Table 2 we can observe that the max value is an outlier. The standard deviation is 98% similar, which gives us the intuition that the dispersion is similar. To consolidate this conjecture of similarity we observe the percentiles: 1% of the samples are at most 1784 ns, 99% similar, percentile 10th are at most 1960 ns, 94% similar, we can verify a suitable similarity when observed until 99th percentile, the similarity is 90% on average. We can conclude that 1% are outliers, therefore a feasible solution.

Table 4 Ffmpeg comparison of regular environment and our emulator

Statics type	HDD	RAM	Emulating RAM
Average time (s)	65.46	3.52	3.58
Coefficient of variation	0.01	0.02	0.09
Frames per seconds	3637	73,311	74,292
Wall clock time (s)	669	48	892

5.4 Evaluation of a user space application in our emulator

Our concern about the emulator is the performance and the time distortion of a user space program. In this section, we evaluate a simple userspace program, in a real environment compared with our emulator. We picked a common tool that handles videos, the *ffmpeg* [Fast Forward MPEG (Motion Picture Experts Group)], this tool is CPU bound. Our target is to convert the video, this task implies read some chunks of the video (read I/Os), convert the video (CPU bound) and write the converted video (write I/O).

For this experiment, we picked a random video with 3.5 GB in size and a total of 229,838 frames. The conversion from MP4 (MPEG Layer-4 Audio), original video format, to h264 (Hikvision 264) makes uses of several resources of the target architecture, which generates some workload. The experiment was done using a RAM disk as the storage backend, giving us a reference point to our experiment. We picked this block device due to the well-known behavior, as observed in the Sect. 5.1. In this manner, we decreased the entropy that may arise. The *freezing time* emulator parameters were set to simulate the RAM storage device, but using the HDD as the storage backend. So we evaluated the experiment in the regular environment, then in our emulator. To validate the experiment we executed ten times in each environment and the results can be seen in Table 4.

Results in Table 4 show that we were able to mimic the RAM storage backend behavior. The accuracy is 98% on average, it is really close to the time elapsed to process the conversion of the video using the RAM storage backend. These values can be confirmed by the same value of the Frames per Second (FPS), which is also 98% as expected. According to the coefficient of variation, the low value indicates that the accuracy of the ten runs was enough to validate the results. The overhead of the emulator compared to the wall clock was just 223 s (25% of the real machine time), when we are emulating the RAM storage. Notice that, when using simulators this overhead would be much greater [e.g. the simulation time would take a thousand times more (Sanchez and Kozyrakis 2013)].

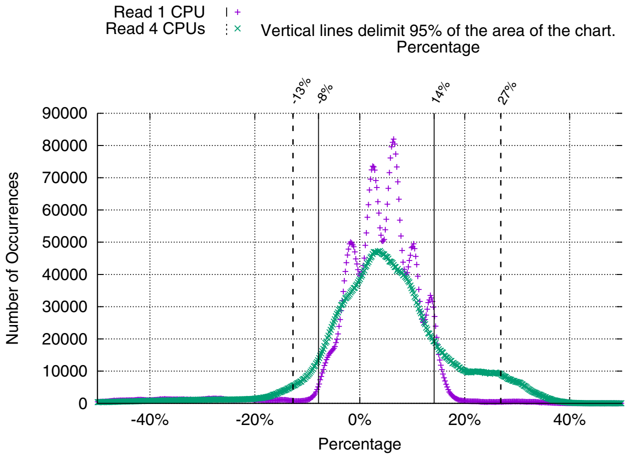


Fig. 9 Rate of the differences of I/Os between the SSD device and the same device emulated. Emulating with one and four vCPUs. Read operations only

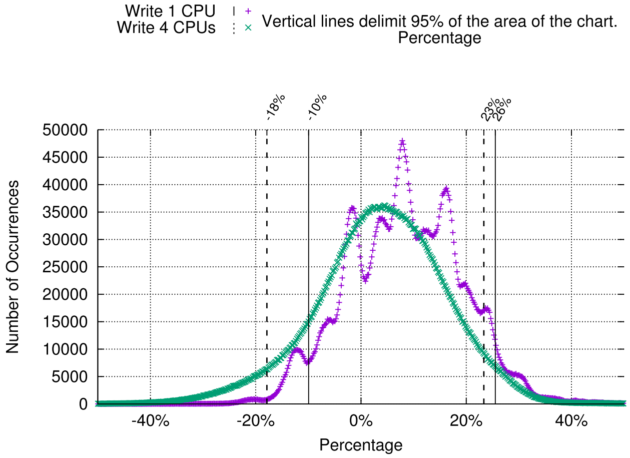


Fig. 10 Rate of the differences of I/Os between the SSD device and the same device emulated. Emulating with one and four vCPUs. Write operations only

5.5 Overhead of the emulator

As described in Sect. 4.4, the overhead of the emulator relies mostly on the mechanism of *kicking* the guest, this occurs because it is not instantaneous and cannot be run in parallel, which means that with an increased number of vCPUs we have increased overhead. The next chart shows the overhead of the emulator with the increase of vCPUs.

Figure 9 represents how many I/O requests were faster or slower than the default behavior of the SSD. The representation on the x-axis are: negative values are how much faster the I/O request was, positive values how much slower the I/O request was. The Y-axis indicates how many I/O requests occurred. Each type of I/O request is limited by the vertical lines where 95% of the samples are. The intervals

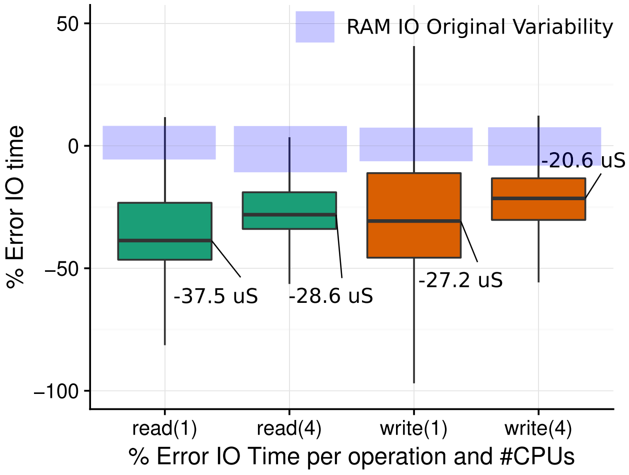


Fig. 11 Evaluation of overhead when using the emulator with delay 0, in a HDD backend, compared to RAM. Overhead goes from 37.5 to 20.6 µs in median

with one vCPU are represented by a continuous vertical line and symbol +, while four vCPUs are dashed vertical lines and symbol *x*.

On both Figs. 9 and 10 the curves are slightly offset from center (0%), this means that the value we chose to simulate the SSD was not accurate enough, setting a smaller value should just offset the curves to the center. Also, the precision of the emulator is about 80% of the real behavior although previous tests show higher precision on average.

On the other hand, the overhead using an HDD backend to simulate a RAM device (which should be the worst case) is shown in Fig. 11. We can see how the overhead goes from 37.5 to 20.6 µs in absolute terms. On percentage-wise, this overhead may seem big, but the absolute time is small compared even with the usual RAM variability observed.

Section 5.3 presents a new approach to emulate a device that does not exist in the machine that emulation would take place. The method moves the clock back in time and possibly could insert some entropy on the latencies. However, an evaluation of the user-space process already has been done in Sect. 5.4. Nevertheless, we agree that a higher depth overhead analysis would bring more insights. We consider such a broader evaluation as future work.

6 Conclusions

Recently, a large number of new storage devices have been proposed. Being able to emulate these devices in real conditions is quite useful. However, the emulation of these devices is challenging because their performance is superior to the available back end. In this work, we demonstrated the use of an emulator that uses a time manipulation technique called freezing time. This technique was implemented using the emulator for virtual machines *QEMU*. The code of the emulator is available for free under GPL license on (Elias 2019).

Among several experiments present, we were able to emulate an SSD using a regular HDD obtaining accurate results. The average latency observed with the emulated device was only 7% lower than those observed in the validation with a real device for both read and write I/O operations. An additional experiment with a newer device, the Intel DCPMM, also showed promising results. Future work includes conducting evaluations using macro benchmarks to verify that the results on the emulated and real devices show compatible results.

In addition to the accuracy of the emulation, it is important to note that the proposed approach only requires modification of the hypervisor code. The guest's operating system is unaware of the emulation and for the measurement tools the existence of the device emulation is completely transparent. Besides, it was demonstrated that our implementation has a small overhead and is feasible in practice.

Acknowledgements This work was partially supported by the Spanish Ministry of Science and Innovation under the TIN2015-65316 Grant, the Generalitat de Catalunya under contract 2014-SGR-1051, the Serrapilheira Institute (Grant number Serra-1709-16621), as well as the European Union's Horizon 2020 Research and Innovation Programme, under Grant Agreement no. 671951 (NEXTGenIO) for the extensions added after the MASCOTS paper.

References

Amsden, Z.: Timekeeping virtualization for x86-based architectures. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/virtual/kvm/timekeeping.txt (2010a)

Amsden, Z.: Timekeeping virtualization for X86-based architectures. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/virtual/kvm/timekeeping.txt. Accessed 15 Apr 2015 (2010b)

Axboe, J.: blktrace user guide. http://www.cse.unsw.edu.au/aaronc/iosched/doc/blktrace.html (2007)

Bayati, M., Bhimani, J., Lee, R., Mi, N.: Exploring benefits of NVME SSDS for bigdata processing in enterprise data centers. In: 2019 5th International Conference on Big Data Computing and Communications (BIGCOM), pp. 98–106 (2019). https://doi.org/10.1109/BIGCOM.2019.00024

Bhavik, S.I.: (Ahmedabad (2016) Methods and systems for performing a read ahead operation using an intelligent storage adapter. http://www.freepatentsonline.com/9311021.html

Bona, L.C., Elias, A., Ziviani, A.P., Cortes, T., Nou, R., Alves, M.A.: Freezing time: a new approach for emulating fast storage devices using VM. In: 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), IEEE, pp. 16–24 (2018)

Elias, A.: Emulator source code. https://github.com/alessandro11/qemu-freezing-time.git (2019)

Flower, J., Gajjar, K.: Adaptive intelligent storage controller and associated methods. US Patent 9,256,542 (2016)

Gordon, A., Har'El, N., Landau, A., Ben-Yehuda, M., Traeger, A.: Towards exitless and efficient paravirtual I/O. In: Proceedings of the 5th Annual International Systems and Storage Conference, ACM, New York, NY, USA, SYSTOR '12, pp 10:1–10:6 (2012). https://doi.org/10.1145/2367589.2367593

Gu, Z., Zhao, Q.: A state-of-the-art survey on real-time issues in embedded systems virtualization. J. Softw. Eng. Appl. **5**(04), 277 (2012)

Gupta, D., Yocum, K., McNett, M., Snoeren, A.C., Vahdat, A., Voelker, G.M.: To infinity and beyond: time warped network emulation. In: Proceedings of the twentieth ACM symposium on Operating systems principles, ACM, pp. 1–2 (2005)

Hajnoczi, S.: Coroutines in QEMU: the basics. http://blog.vmsplice.net/2014/01/coroutines-in-qemu-basics.html (2014)

Kim, J., Ahn, S., La, K., Chang, W.: Improving i/o performance of NVME SSD on virtual machines. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, ACM, pp. 1852–1857 (2016)

Kim, T.Y., Kang, D.H., Lee, D., Eom, Y.I.: Improving performance by bridging the semantic gap between multi-queue SSD and I/O virtualization framework. In: 2015 31st Symposium on Mass Storage Systems and Technologies (MSST), IEEE, pp. 1–11 (2015)

Knuth, D.E.: The Art of Computer Programming, Volume 1 (3rd Ed.):: Fundamental Algorithms, pp. 193–200. Addison Wesley Longman Publishing Co., Inc, Boston (1997)

Lee, Y.C., Kuo, C.T., Chang, L.P.: Design and implementation of a virtual platform of solid-state disks. IEEE Embed. Syst. Lett. **4**(4), 90–93 (2012)

Lu T, Huang P, He X, Zhang M (2016) Understanding the impact of cache locations on storage performance and energy consumption of virtualization systems. In: 2016 USENIX workshop on cool topics on sustainable data centers

Oh, M., Eom, H., Yeom, H.Y.: Enhancing the I/O system for virtual machines using high performance SSDs. In: Performance Computing and Communications Conference (IPCCC), 2014 IEEE International, IEEE, pp. 1–8 (2014)

Rizzo, L., Lettieri, G., Maffione, V.: Speeding up packet i/o in virtual machines. In: Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on, IEEE, pp. 47–58 (2013)

Russell, R.: Virtio: towards a de-facto standard for virtual I/O devices. ACM SIGOPS Oper. Syst. Rev. **42**(5), 95–103 (2008)

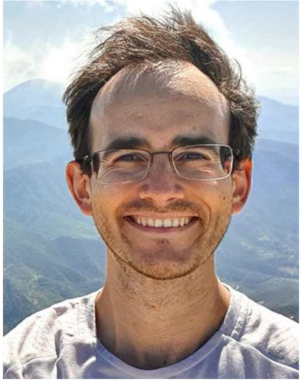
Sanchez, D., Kozyrakis, C.: ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. ACM SIGARCH Comput. Arch. News ACM **41–3**, 475–486 (2013)

Xen_Project (2015) Xen project beginners guide. https://wiki.xenproject.org/wiki/Xen_Project_Beginners_Guide

Luis C. E. Bona is an Associate Professor at Federal University of Paraná, Curitiba, Brazil, where he is member of the Computer Networks and Distributed Systems Lab (LaRSis). He obtained a Ph.D. degreein Electric Engineering at Federal University of Technology - Paraná, 2006, and carried out his postdoctoral studies at the Barcelona Supercomputing Center (BSC), 2013. His research interests include Operating Systems, Computer Networks and Distributed Systems. He acted as coordinator of several research, technological and development projects, both national and international. He also served as chair of the Departmentof Computer Science of Federal University of Paraná from 2008 to 2012.

Alessandro Elias has been working at Lab Center of Computer Science and Free software (C3SL) at Federal University of Paraná, Curitiba, Brazil as Project Manager. He obtained a Bachelor's degree in Computer Science (2017), he is Master Student at LaRSis Federal University of Parana. Worked in a project in partnership at Albert-Ludwigs-Universität Freiburg (2014), researching in distributed storage. Now he is researcher at C3SL.

Andre Purri Ziviani worked at the laboratory Center of Computer Science and Free Software (C3SL) at Federal University of Paraná, Brazil as a researcher, graduated at the same institution in 2016.



Ramon Nou has been working at BSC on the Autonomic System and e-Business Platforms group of the Barcelona Supercomputing Center (BSC) until 2008 when he switched to the Storage-system group at BSC since 2009 as a researcher. In 2008, he obtained his Ph.D. with the topic "Using online simulation to improve QoS on middleware". Ramon has a wide view on all computer levels, with expertise in optimization, performance measurements and simulation/modelling of complex systems. Now is co-leader of the Storage Systems for Extreme Computing research group since January 2019.

Toni Cortes is an associate professor at Universitat Politècnica de Catalunya (since 1998). He received his M.S. in computer science in 1992 and his Ph.D. also in computer science in 1997 (both at UPC). Currently he develops his research at the Barcelona Supercomputing Center, where he acted as team leader from 2006 till 2019. His research concentrates in storage systems, programming models for scalable distributed systems and operating systems. He has published 118 technical papers, 3 book chapters, and has co-edited one book on mass storage systems. In addition, he has also advised 15 PhD thesis.

Marco A. Zanata Alves is an adjunct professor and researcher at the Department of Informatics, Federal University of Paraná (UFPR), Curitiba, Brazil (since 2016). He graduated in Computer Science at UNESP (2006). Obtained Master and a Ph.D. degree in Computer Sciences from UFRGS (2009 and 2014). Worked as a postdoctoral researcher at UFRGS between 2014 and 2016. Has experience in Computer Science, focusing on the Computer Architecture area.