2018 IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems

Freezing Time: A New Approach for Emulating Fast Storage Devices Using VM

†Luis C. E. Bona, †Alessandro Elias, †Andre P. Ziviani, ‡§Toni Cortes, ‡Ramon Nou, †Marco A. Z. Alves
†Federal University of Paraná - Curitiba, Brazil
‡Barcelona Supercomputing Center - Barcelona, Spain
§Universitat Politècnica de Catalunya - Barcelona, Spain
E-mail:†{bona, aelias, ziviani, mazalves}@c3sl.ufpr.br, ‡{toni.cortes, ramon.nou}@bsc.es

***Abstract*—recently we are seeing a considerable effort from both academy and industry in proposing new technologies for storage devices. Often these devices are not readily available for evaluation and methods to allow performing their tests just from their performance parameters are an important tool for system administrators. Simulators are a traditional approach for carrying out such evaluations, however, they are more suitable for evaluating the storage device as an isolate component, mostly due to time constraints. In this paper, we propose an approach based on virtual machine technology that is capable of emulate storage devices transparently for the operating system allowing evaluation of simulating devices within a real system using any synthetic or real workload. To emulate devices in real environments it is necessary to use the currently available devices as a storage medium which creates a difficulty when the device to be emulated is faster than this storage medium. To circumvent this limitation we introduce a new technique called Freezing Time, which takes advantage of virtual machine pausing mechanism to manipulate the virtual machine clock and hide the real I/O completion time. Our approach can be implemented just requiring the hypervisor to be modified, providing a high degree of compatibility and flexibility since it is not necessary to modify neither the operating system nor the application. We evaluate our tool under a real system using old magnetic disks to emulate faster storage devices. Experiments using our technique presented an average latency error of 6.08% for read operations and 6.78% for write operations when comparing a real to device.**

I. Introduction

Datacenter storage requirements have become increasingly complex due to the recent demands imposed by big data processing. These demands have motivated many scientists and companies to propose new ways to store and retrieve data, such as: read ahead operation using an intelligent storage adapter [1], adaptive intelligent storage controller and associated methods [2] and also new storage technologies like NVMe are becoming available, although not widely accessible due to its high cost. Thus, evaluating the impact of these new technologies on data centers by considering the entire stack of operating system storage, workload and real applications is a challenge.

Simulators are important tools for the evaluation of new storage systems. However, due to its time consumption, they are more appropriate for the evaluation of the storage device as an isolated component than a storage system considering all the machine and software layers. In the standard usage case, simulators receive a trace obtained from a system executing a given workload, and deliver a set of performance metrics. This approach can hardly capture the several interactions of a storage system with the various components, such as the core of the operating system. Rarely, the results of a simulation can be extrapolated to the results that could be obtained with a set of workloads.

Running a large number of programs or benchmarks is unfeasible using an offline simulator, an alternative is to use an emulator to make an online simulation. The approach of this paper, is to use such an emulator, based on a virtual machine and dilate the time during IOs. With this emulator, it is neither required to generate a trace nor make changes to programs or benchmarks, since the applications in the guest environment are not aware that devices are not real. An emulator takes time to process the emulated device, and it affects directly the minimum latency and maximum throughput that can be achieved, because for every access the IO has to wait for the emulator and then for the real device. A possible approach is to store the virtual disk entirely in RAM (or external DRAM like device) due to its fast access time. Another approach is to dilate time, meaning that, the time spent inside the guest is not the same as outside of it. We can dilate time linearly for the whole guest e.g. with a factor of ten, the guest would notice one second elapsed for every ten seconds of "real time", this method would distort the time of the overall components.

Inspired by the concept of time dilation, we propose a new technique called Freezing Time, that employs the mechanisms of stopping the virtual machine, on which the emulated device can be faster than the storage available or even devices that do not exist yet, all transparently to the application and almost without any performance limitations. In this paper we present the following main contributions:

- Flexible emulator tool for storage simulation which requires no trace or changes in user space applications or either changes in 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 error of less than 7% considering read and write operations.
- Low overhead tool with less than 25% increase on emulation time, for fast evaluation of new and future storage devices.

2375-0227/18/$31.00 ©2018 IEEE
DOI 10.1109/MASCOTS.2018.00010

- Open source software that is available at [3], under GPL license.

All experiments performed in this paper were made in a reproducible manner (using open-source software and a standard x86 architecture machinery). Our experiments show that we were able to emulate disks with RAM like speeds with an overhead of less than 20% in IO request time while keeping precision as high as 94% on average.

II. Related Work

Lee and Kuo [4] used a RAM disk as a faster storage backend, in this manner they could emulate any device that is slower than the system RAM. The disadvantage of this approach is that RAM is often small and expensive, which makes it impractical for simulating large devices. In our work, we can use any backend to simulate any other storage device of up to RAM speed.

Gu and Zhao [5] addressed this problem disabling all the interruptions and disabling the hardware clocks on the host kernel so that it was partially frozen. This method required to have another machine to emulate the device and process the IOs. Another problem was that the kernel's network driver used an interrupt based system, so they had to develop a new driver using a busy-wait system to communicate with the storage emulator. The option to implement in host's kernel made it extremely dependent on the hardware used to implement.

Gupta et al. [6] explored network time dilation, but it could be used on any other part of the system. The idea was to linearly distort the time seen by the guest by a user defined factor e.g. with a factor of ten the guest would notice one second elapsed for every ten seconds on real time, this method distorted the time of all the components, effectively making every operation take ten times longer. The drawback of this approach is that every component of the virtual machine will have its time distorted and the faster the emulated device the more distortion is required on the whole system, affecting not only the emulated device but every other device. This implementation also makes modifications to the virtual machine monitor (Xen) and the guest's kernel.

III. Background

In this section we are going to present the *Virtio* which is the standard kernel module used by our proposal, explaining its internal architecture. We also present the clock managing for the QEMU used by Freezing Time.

A. Virtual interface for IOs

Emulating a real hardware device costs many CPU cycles, because every behavior of the device has to be emulated. For each access on the emulated device each request must be interpreted and act according to the real device, causing an increased load on the host machine and lowering throughput and consequently, IOPS (Input/Output Per Second) [7]. *Virtio* (Virtual Input/Output) was created in a way that the guest and host could communicate without having to simulate a device or having to *kick* the guest, therefore lowering the load on the host and increasing throughput and IOPS [8]. It was also designed with compatibility in mind so that it would not be necessary to make big changes on the guest and the VMM. *Virtio* presents itself to the guest as a PCI device, this way the guest only needs to implement a new PCI driver, and VMM need only add vring (Virtual Ring) support to the devices they implement [7].

> The target to reach with *Virtio* (Virtual Input/Output) is to unify how the probe in the Linux Kernel occurs, so different implementations can be developed, to be supported by hypervisor A or B. So the step to reach this target, is to guide towards uniformity to provide a common ABI (Application Binary Interface) for general publication and use of buffers. Deliberately, our *Virtio_ring* implementation is not at all revolutionary: developers should look at this code and see nothing to dislike [9].

The *Virtio* driver is implemented as a stack, transport and configuration. The desired goal is to reduce the duplication code in virtual device drivers, so abstraction is mandatory. To achieve the abstraction, *Virtio* is provided with a set of common helpers which virtual drivers can use. The task is to create a transport abstraction for all virtual devices which is simple and close to optimal for efficient transport.

The *probe* function from the driver is called when suitable *Virtio* device is found. The configuration happens in four steps: reading and writing feature bits, reading and writing the configuration space, reading and writing the status bits and device reset. The device looks for device-type-specific feature bits corresponding to features it wants to use, such as the VIRTIO_NET_F_CSUM feature bit indicating whether a network device supports checksum offload. Feature bits are explicitly acknowledged: the host knows which feature bits are acknowledged by the guest, and hence which features that driver understands. The second step is the configuration space, a structure associated with the virtual device containing device-specific information. This structure can be both read and written by the guest. These mechanisms give us room to grow in future, and for hosts to add features to devices with the only requirement being that the feature bit numbers and configuration space layout be agreed upon. There is also a status word (8 bits) which the guest uses to indicate the status of the device probe; when the VIRTIO_CONFIG_S_DRIVER_OK is set, it shows that the guest driver has completed feature probing. Finally, reset operation is expected to reset the device configuration and status bits [9].

The *virqueue* has an API *find_vq* that populates the structure for the queue, giving the *Virtio* device an index number. A *virtqueue* is simply a queue into which buffers are posted by the guest for consumption by the host, and multiple buffers can be added for batching, improving performance since the cost to notify the host is expensive.

The *Virtio_ring* is the transport for Linux *Virtio*, consists of three parts: the descriptor array where the guest chain contains length/address pairs, the available ring that indicates

which descriptor chains are ready for use and the used ring where the host indicates which descriptor chains were used. The size of the ring is variable but must be a power of two. Each descriptor contains the guest's physical address of the buffer, its length, an optional next buffer for chaining and two flags: one indicates if it is valid and another if it is writable or readable. The available ring consists of a free-running index, an interrupt suppression flag, and an array of indices into the descriptor table (representing the heads of buffers). The separation of the descriptors from the available ring is due to the asynchronous nature of the *virtqueue*: the available ring may circle many times with fast-serviced descriptors while slow descriptors might still await completion. Used ring and available ring are similar; they are written by the host as descriptor chains are consumed. The flags that indicate if it is a used or available buffer are used for optimization since notification forces the guest to exit from the guest mode, those flags are also used by the guest driver to advise that further interrupts are not required.

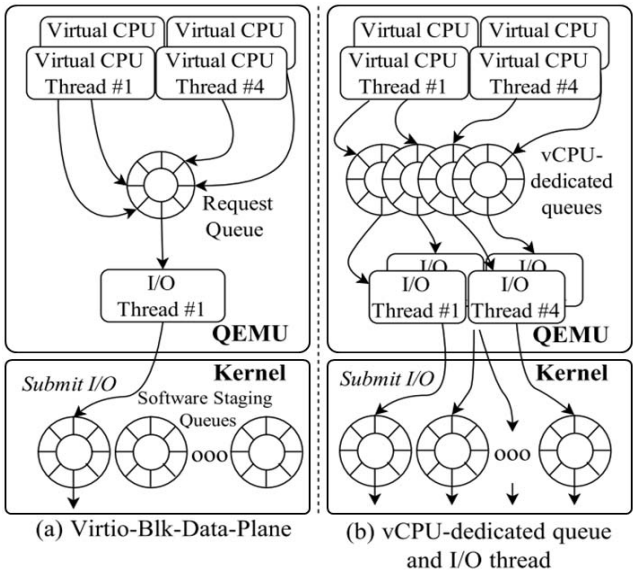


Fig. 1. IO path in the Virtio-queue, from VCPU until the device block [10].

Figure 1 represents the path the IO take in the system from the VCPU to the device. *Virtio* block is a part of the *Virtio* system 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.

B. Timekeeping: choosing a clock source

Emulating a clock source decreases performance but it increases the guest compatibility, so most, if not all, of the VMMs have an option to do so. Common options are HPET (High Precision Event Timer) [11] and TSC (Time Stamp Counter) [11]. TSC is a good clock source to use on the host because there is independent hardware in the CPU with its dedicated circuitry which is not affected by CPU clock changes. However, there are some drawbacks when sharing it with a VM, since it is the same clock as the host, 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.

Another problem is live migration; some VMM have an option to migrate a running guest to a different host without powering the guest off. During the migration, the guest needs to disable interruptions and, during this period, time may need to be caught up. After live migration, timers based on the TSC or HPET (if it is not emulated) may be running at different rates requiring some adjustment by the VMM. Additionally, if the destination host has a faster TSC it cannot be exposed to the guest without the potential of time running faster than normal, a slower TSC is less of a problem as the VMM can make adjustments to make it catch up with the source host TSC [11].

Kvmclock was explicitly designed to solve those problems, the guests can register a memory page to contain the *kvmclock* data and the VMM will write to it until explicitly disabled or the guest is turned off. Since it is neither emulated nor the host's TSC the VMM will write multipliers and offsets compared to the host's TSC so the guest can convert those values back into nanosecond resolution seeing only the time it was running with a small overhead. *QEMU* wraps those features as functions by the name KVM_GET_CLOCK and KVM_SET_CLOCK, and those are used on live VM migration.

We managed to use those features to mimic a guest live migration making the time taken to complete the VM disk IO controllable. When the VMM receives an IO request, it will *kick* the guest and save its clock. After the IO has finished, the VMM restore the clock and resume the guest, making it believe that specific time was spent on this IO.

IV. Freezing Time Storage Emulator

Our main idea behind Freezing Time was to build a fast and efficient disk emulator using KVM and *Virtio* that would only dilate the time of the emulated device and not the whole system, allowing the user to make time comparisons and benchmark of full systems. We also wanted to create a tool that did not require the application to be recompiled and would not increase the total time of the experiment in orders of magnitude as in cycle-accurate simulators [12], [13], [14].

Our approach was to detect the guest IO as early as possible inside the VMM and resume the guest just before guest's VCPUS returning to guest mode (we will call this event by its syscall name *KVM_RUN*).

The nomenclature *KVM_RUN* is used by *Kernel-based Virtual Machine* (KVM), among other nomenclature such as *KVM_CREATE_VM*, *KVM_CREATE_VCPU*, however in our context the most important is *KVM_RUN*, which consist in setup the CPU in guest mode, so the VCPU can run each instruction natively. Each VMM that wishes to run on behalf of KVM must use those APIs, in our case QEMU uses it. So tacking off the CPU from *KVM_RUN* means kicking

the VCPU from guest mode, and the control goes back to QEMU. On this manner we have a chance to manipulate the VM_CLOCK, since it is not running. With this approach, it becomes possible to precisely emulate new devices with a small overhead.

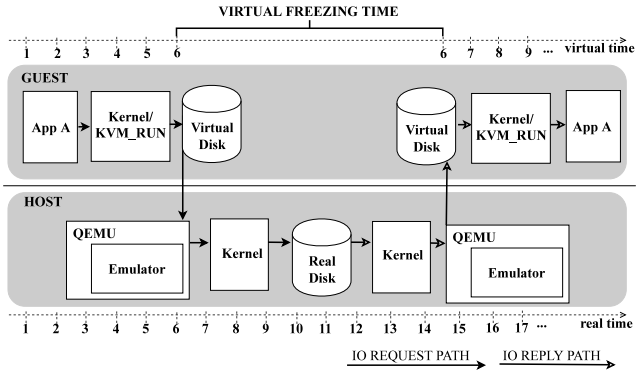


Fig. 2. Flow of an IO request until completion, from guest to the device in the host (the time unit are meaningless, it is just a reference when the guest is "frozen").

Figure 2 represents an abstraction of the implementation that we propose in this paper which is transparent to both host's and guest's kernel. The lower half of the figure represents the host system and the Time-line perceived by the real machine, while the upper half is the virtual system (guest) and the virtual Time-line perceived by it, time units in this figure are meaningless, just to show when the guest's time is "frozen".

At the point of guest time number one, the guest user space process requests an IO to the guest's kernel, after that the kernel converts it to a disk command and issues it to the virtual disk. *QEMU* then receives this command and *kick* the guest at the number seven; the VMM translates the command to an IO request to the host's kernel, which in turn sends the command to the real device. Once the device processes it, it forwards the reply to *QEMU* through the host's kernel, so at this time *QEMU* can send the reply to the emulator. After the emulator has finished processing the reply, *QEMU* issues a *KVM_RUN* and send it to the virtual disk in the guest. Finally, the virtual disk has the reply and sends it to the process that requests it through the guest's kernel at guest's time number eight. Notice that we could even inject time in the virtual machine in order to simulate some specific behavior of the disk device.

With our approach of pausing the time in the guest as soon as the IO is detected (eventually injecting time) and resuming the VCPUS as soon as it goes into context, we make the guest believe that the time has not elapsed, as shown in the gap from host time number seven until fourteen on Figure 2.

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

A. Implementation

Our emulator is based on *QEMU*, of which at the time of the implementation the latest version was 2.5.0. Implementations on other VMMs systems besides QEMU can be studied, but as we need to modify the inner layers VMMs' source code should be available. The first challenge was understanding the inner working of *QEMU* and its interaction with KVM. *QEMU* usually has the following threads: one thread per vcpu, one *iothread* and some other helper threads (e.g. VNC) that are not relevant in this context. Every time the VMM has to do a privileged operation (e.g. access the back storage) it has to *kick* the guest and lock a global mutex, serializing every IO, when this operation finishes the IO the thread releases the mutex and issue a *KVM_RUN* to resume the guest operation [8], [7].

That implementation creates a big impact on performance as for every IO, not only the guest has to be *kicked* but it also serializes every access. Since version 1.4, *QEMU* has a feature called *dataplane*, and the idea is that a device configured with this feature will have its *iothread* therefore not being bottlenecked by the global mutex. Another advantage to this feature is that by creating a new thread for this device, given that the host has enough CPU cores, the load on the machine will be more evenly distributed among the CPUs, thus decreasing latency.

Our implementation works on both *iothread* and *dataplane* modes, but the focus is on the *dataplane* mode due to its lower latency giving more precise results in our emulator. The main concern was to detect, as soon as possible when an IO occurred on the emulated disk to *kick* the guest and set, as late as possible, the guest clock and issue a *KVM_RUN* after the IO has finished.

The *dataplane* thread will keep polling the FD (File Descriptor) which represents its device, so we inserted our code right after the thread identifies that an IO request is *popped* from the *Virtio_ring*, and not a command to the device itself (i.e. restart the device). Thus, our code request the global mutex lock, to avoid the *iothread* or VCPUS threads from trying to do a privileged instruction and interfere with the process, *kick* the guest and save the guest clock. Now that the guest is out of context the time elapsed from now on will be undetected by the guest, which means that the host can serve the requested IO on any medium and the guest (when resumed) will notice only the user-defined latency.

After the IO has finished, we synchronize the *QEMU* threads just before *KVM_RUN*, and the global mutex lock is released so that the other threads can finish their tasks. The *dataplane* thread stays on a busy-wait for the other threads, and when they arrive at the barrier it sets the clock back to when the guest was paused, lifts the barrier and the guest returns to *KVM_RUN* mode.

B. Virtual IO path: From the guest application to the host storage

Every IO made by the guest has to go through multiple layers from guest's virtual memory, guest's kernel, *QEMU* and host's kernel before eventually reaching the storage backend. Figure 3 shows a high-level abstraction of the path an IO goes through. This section we will explain these steps in more detail.

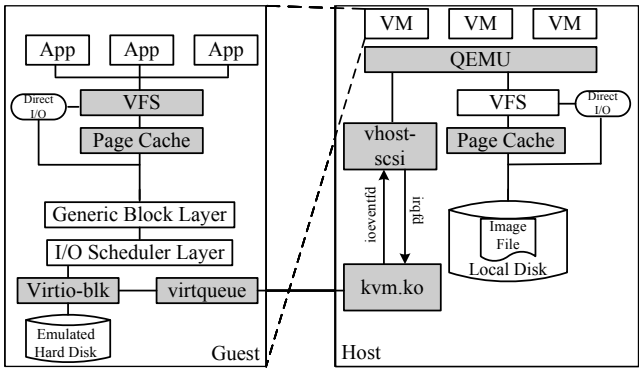


Fig. 3. Going through in all layers of an IO request. Adapted from [15].

Whenever a process running in the guest generates an IO, the guest's kernel checks if that request is already on the page cache, like a normal non-virtualized system would, in the case of a page cache hit, then the kernel returns the requested data. Now, let's suppose that the IO is neither on the guest cache nor in the host's, then, the guest's kernel sends the request to the generic block layer which in turn sends to the IO scheduler, then to the *Virtio* block driver and finally to the *virtqueue* of the emulated device. Now on the host side, the KVM kernel module detects the request and sends it to the *vhost-scsi* inside the *QEMU*, explained in more detail below. Then the *dataplane* thread wakes up and begin to process the request, from now on the host's kernel recognizes that thread as a normal process and treats the request as it would for any other process in the host going through the VFS and page cache all the way to the local disk.

After the host's kernel finalizes the request and notifies the *QEMU* process, the guest's kernel polls the PCI bus and retrieves the answer to the request and forwards it to the guest's process.

C. Overhead of our emulator

The time seen by the guest during the IO is ideally equal to zero but we cannot detect an IO and *kick* it instantly. The overhead is composed of two parts: the bigger one is the time elapsed since the guest kernel issued the request to the *Virtio* driver and the *dataplane* thread detects a request and *kick* the guest. The second part is when the IO is finished, the clock is restored in the guest and the VMM issues a *KVM_RUN*. During the injection or when the guest is frozen, interruptions in it are not affected, since we just apply time dilation after all VCPUs are out of context.

When a *QEMU* device is configured to use the *dataplane* mode, the thread polls only that FD (File Descriptor that have been setup to the *Virtio* device) which means that it will do fewer checks since it already knows what device is represented by that FD. When the guest adds a request to the *Virtio_ring* it generates an interruption on the *IRQ* of the device, which the host kernel translates to an event on the FD. The *dataplane* thread removes the request from the *Virtio_ring*, at this time we request the global mutex lock to avoid the *iothread* or VCPUS threads from trying to do a privileged instruction and interfere with the process, checks if it is a data (read or write) request and, if it is, save the guest clock and *kick* it. The label *A* on Figure 4 represents the cost of *kicking* all the VCPUS, this process is serial and not instantaneous, so the time taken between *kicking* the first and the last vcpu is distorted. This time distortion directly affects the overhead of the emulator, decreasing the maximum achievable performance. After *QEMU* finishes *kicking* the guest, the host can complete the request without the guest noticing the time passing by.

QEMU uses a technique called *coroutine* [16] to try to mitigate the problem with multiple call-back functions [17]. We modified this technique to include our injection mechanism after the IO is finished, we force the coroutine to sleep for a user-defined amount of time which will be seen by the guest as the device's latency. The timer starts after the *KVM_RUN* command.

At the end, we synchronize the vcpu threads just before issuing the *KVM_RUN* command and restore the guest clock, label *B* on Figure 4 represents the overhead of the VCPUS returning to execution.

In the following section, we will validate the procedure described above, by acquiring statistics through tracing tools and synthetic IOs.

V. Experimental Results

In order to show the emulator effectiveness we did the following set of experiments: First, in the section V-A, we show the fastest IO that technically could be achieved in our test system using a RAM disk device. Following that, in section V-B, we show the results of our experiment of emulating an SSD without any modifications to guest nor the host (except on *QEMU*) using a slower storage backend. Finally, in section V-D we measure the overhead of the emulation and how long the guest thinks the experiment ran and how long it took.

The experiments in this paper were dedicated to simulating an SSD. At the moment of the experiments described in this paper, the devices available were: Server Grade SSD Cloud Speed 500 (model TG32C1) manufactured by Smart Storage Systems. HDD (Hard Disk Drive) (model JPT39C), with size of 1 TB, using SATA interface, speed of 3.0 Gb/s, and 7200 RPMs manufactured by Hitachi Global. The selection was done to cover a mechanical device (HDD) and a non-mechanical device (SSD). The host's and guest's operation system 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(tm)-6300 Six-Core Processor, 3.5 GHz, with 12 GB of DDR3. All the experiments executed in this paper were performed on this specific system mentioned above. Thus, we expect that values obtained in our experiments will vary from system to system.

We made the experiments using *blktrace* (Block IO layer tracing), *blkparse* (Block IO layer parser) and *fio* (flexible IO

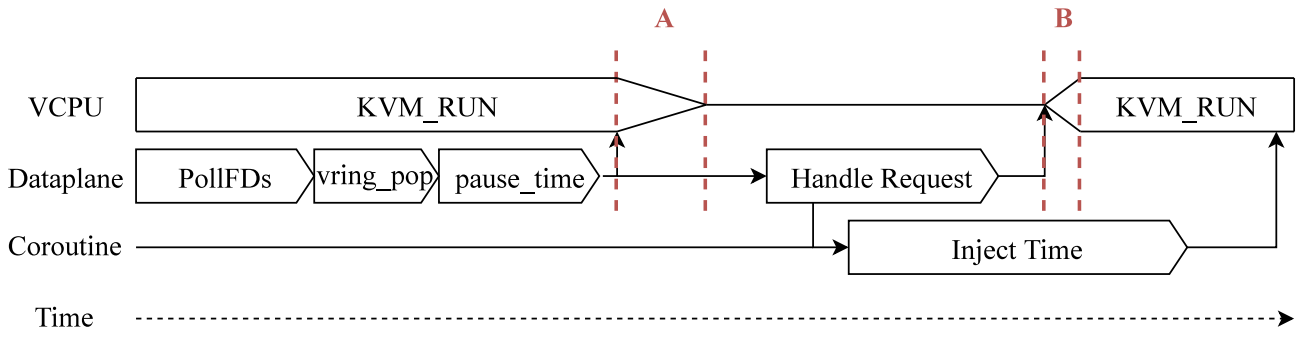


Fig. 4. Overview of the time dilation mechanism inside the hypervisor (*QEMU*).

tester). *Blktrace* is a block IO layer tracing utility that provides the ability to collect detailed traces from the kernel for each IO processed by the block IO layer [18]. *Blkparse* parses the output events stored in files generated by *blktrace* in a human readable way [18], while *fio* simulates a specific workload configured by the user such as sequential or random read/write, block size, number of threads, etc.

QEMU has two types of storage backend, *iothread* and *dataplane* [19]. Our tests showed that the *dataplane* mode is one order of magnitude more efficient than the *iothread* mode, due to this fact, only the *dataplane* mode was used in our evaluation.

The experiments consist of the following steps:

1) Run *blktrace* to collect IO events. We are only interested in the response time.
2) While *blktrace* is running, run the workload; in our experiments, *fio* plays that rule, making IOs to the device in question.
3) When the workload has completed, stop the *blktrace* utility (thus saving all traces over the entire workload).
4) Extract the pertinent IO information from the traces saved by *blktrace* using the *blkparse* utility.

The experiments consist of running the *fio* program five times on the SSD, HDD and RAM devices described above, synchronously reading and writing, data with size of 4 GB with chunks of 4 KB to the backend storage device, for one and four VCPUS with a fixed amount of 2 GB of RAM in the guest. After each execution, the guest cache was flushed, and *QEMU* was configured to not cache IOs on the host.

A. Empirical IO speed limit

This section describes our emulator limit in performance, based on our experiments, and how we achieved it.

To illustrate the optimal performance, we configured *QEMU* to use the RAM as the storage backend to show the fastest IO that can be achieved, and show some CDF (Cumulative Distribution Function) charts with the results.

B. Emulating an SSD

Now we will show how the HDD and SSD react to the experiments without the emulator, as we did with RAM, later

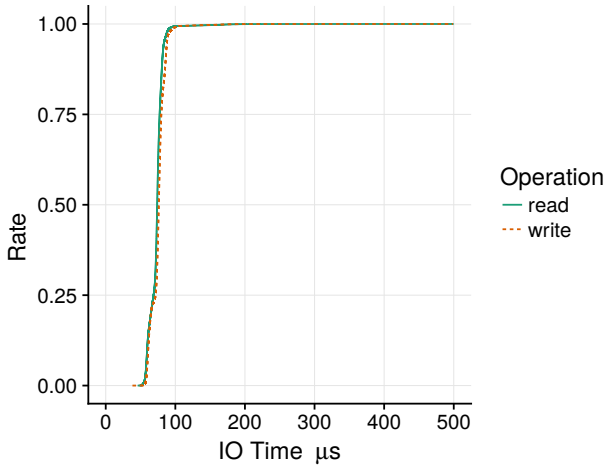


Fig. 5. Cumulative distribution using a disk in RAM as backend, technically, fastest IO that can be reach.

we will simulate the SSD but using a HDD as storage backend. The injected time was obtained empirically, by running the experiments on the SSD without any virtualization and injecting this time on each IO.

Figure 6 shows how the HDD reacts to the tests inside the guest without any modification in *QEMU*. Figure 6 shows the CDF (Cumulative Distribution Function) of the time to complete an IO request on an HDD, with that plot we can see that below 0.125 (or a 12.5%) of the write requests are very fast due (before 2000 µs) to the HDD buffer. As we can see, 100% of write requests are below 5 ms and reads are between 2.5 ms and 10 ms. It is clear that samples on the HDD are multiple orders of magnitude more heterogeneous than the RAM or an SSD.

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

Figure 8 shows the results of the experiments using our emulator to simulate the SSD using the HDD storage backend. The aim of Figure 8 is to show how our technique provides

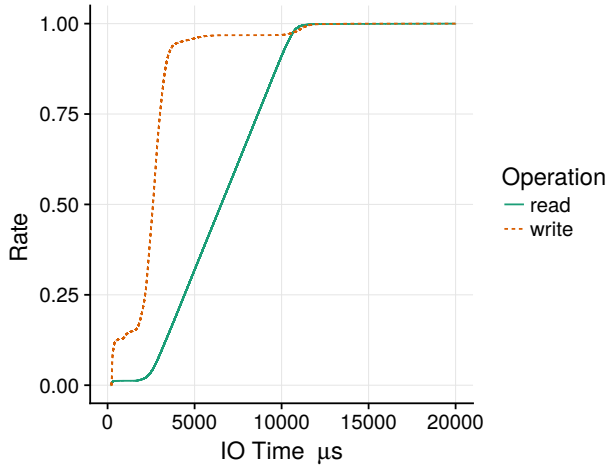


Fig. 6. Cumulative distribution using HDD as backend with emulator off; 100% of write requests are below 5 ms and reads are between 2.5 ms and 10 ms.

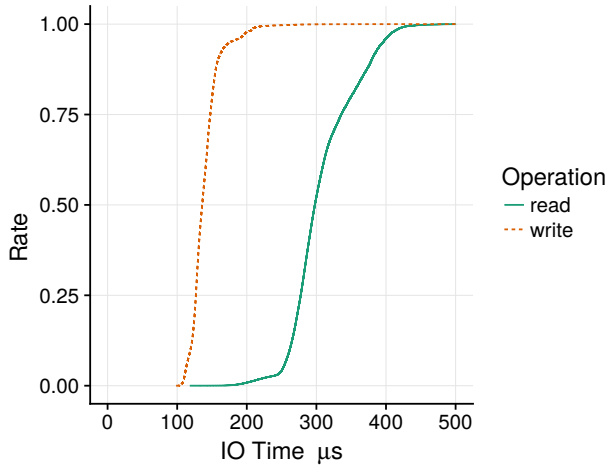


Fig. 7. Cumulative distribution using SSD as backend with emulator off; 100% of write requests are below 200 µs and reads are between 200 µs and 400 µs.

results inside the observed variability of the original device. The results are divided by operation and by the number of VCPUs used and each boxplot shows the distribution of the error difference between the mean SSD request and each of the IO requests of the emulated SSD with the HDD backend (for each scenario). The original variability of the IO 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. These results are obtained using the mean of the IO requests as input for the delay parameter, that is the simplest way to model, but more complex "delay" simulations could also be used. For example, we can include cache effects in the simulation.

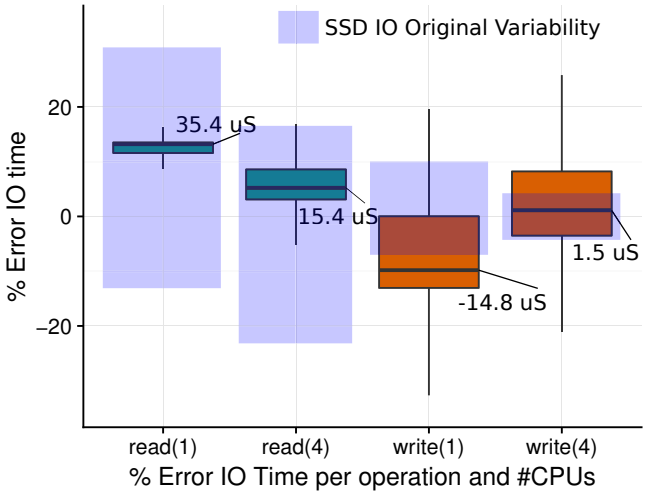


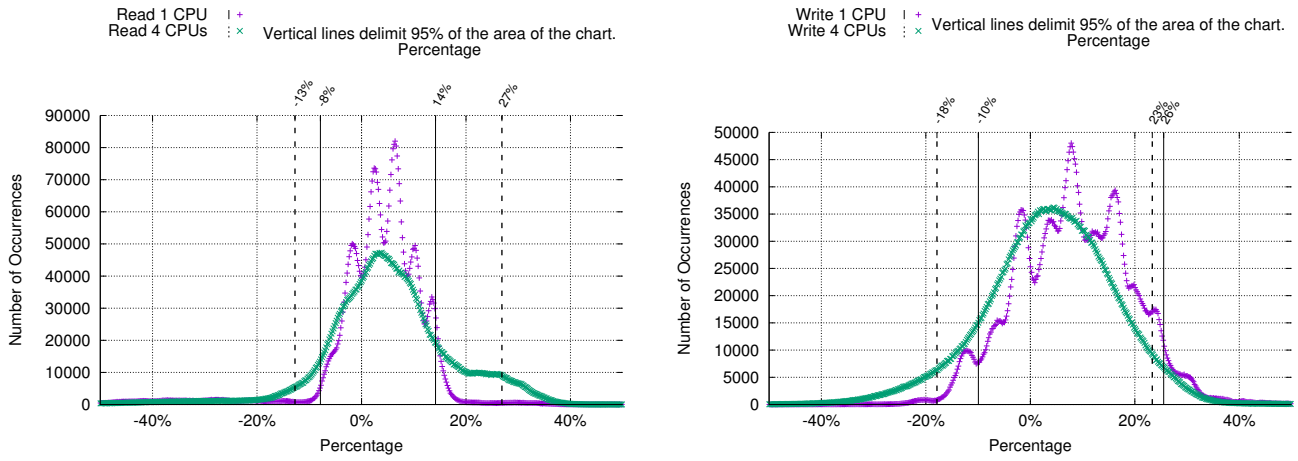
Fig. 8. Experiment emulating the SSD using an HDD as storage backend. The original variability of the IO requests in the SSD is shown with a transparent shaded rectangle, as we can see the emulation is inside the rectangle on most of the scenarios, although we have used the mean of the IO requests as input for the delay parameter.

C. Performance evaluation of a user space application in our emulator.

A question that may arise is, how good is the performance of user space process, using our emulator? What we want to show in this section is the performance of a simple application, in a regular environment compared with our emulator. To show the performance, we chose the *ffmpeg* (Fast Forward MPEG (Motion Picture Experts Group)), since is a simple tool which is IO and CPU bound. Our objective is, to convert the video, which implies read some chunks of the video (read IOs), convert the video (CPU bound) and write the converted video (write IO).

To accomplish this task we picked a random video with 3474123501 bytes in size, and 229838 frames. The original format of this video is mp4 (MPEG Layer-4 Audio) which we convert into h264 (Hikvision 264). So, to evaluate and get the results we setup a RAM disk as the backend storage device, as a reference to our experiment. The reason that we chose this device is because of its well-known behavior, as seen in section V-A, on this manner we eliminated any entropy, that could interfere with the results. In our emulator we setup the parameters in the *freezing time* layer to emulate this RAM storage device, but using the HDD as storage backend. Then we ran the conversion of the video in the regular environment, then in our emulator. To validate we run ten instances in each environment and the results can be seen on the Table I:

Analyzing the results on the Table I we observe 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 value can be confirmed by the same value of the FPS, which is also 98% as expected. According to the coefficient of variation the low value indicates that the



(a) READ (b) WRITE

Fig. 9. Rate of the differences of IOs between the SSD device and the same device emulated, The emulator with one and four VCPUs.

TABLE I
PERFORMANCE OF FFMPEG, COMPARE THE REGULAR AND OUR EMULATOR ENVIRONMENT.

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	73311	74292
Wall clock time (s)	669	48	892

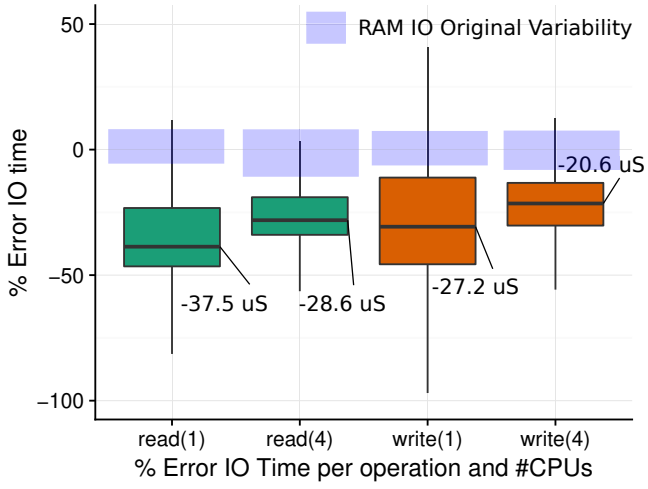


Fig. 10. Overhead when using the emulator with delay 0, in a HDD backend, compared to RAM. Overhead goes from 37.5 µs to 20.6 µs in median.

accuracy of the ten instances were enough to validate the results. The overhead of the emulator compared to the wall clock, is just 223 seconds (25%), 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 1000× more than an virtual machine [20]).

D. Overhead of the emulator

As explained earlier, the overhead of the emulator relies mostly on the fact that the process of Freezing Time using the *kick* mechanism 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 IO requests were faster or slower than the default behavior of the SSD. Positive values in X-axis represent how much slower the IO request was (in percentage), and negative values are how much faster the IO request was. The Y-axis indicates how many IO requests occurred. The vertical lines limits each type of IO request, where 95% of the samples are. The experiment using one VCPU is represented by the $+$ symbol and continuous vertical line, and the x symbol and dashed line represents the experiment using four VCPUs.

In both Figures 9 (a) and (b), the curves are slightly offset from center, 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, but previous tests show that, on average, they are similar.

On the other hand, Figure 10 shows the overhead when using an HDD backend to simulate a RAM device (which should be the worst case). We can see how the overhead goes from 37.5 µs to 20.6 µs in absolute terms. On percentagewise, this overhead may seem big, but the absolute time is small compared even with the usual RAM variability observed.

VI. CONCLUSION

In this paper we presented a solution to emulate existing and non-existing disk devices. The motivation to provide such environment is to emulate expensive and fast devices to make decisions in infrastructure. Nevertheless it could also be used for fine tuning IOs in highly demanding applications or any

other purpose that needs to tune latencies. We achieved this goal by implementing the emulator using a time dilation technique. Our implementation indeed has an overhead, but we showed that it was small and performance impact was also small due to the *Virtio* framework. Another goal that we managed to accomplish was to be able to inject a specific amount of time to each IO. Our main desired feature of not making any modifications to neither guest's kernel nor host's kernel was also accomplished.

We implemented the time dilation mechanism by *kicking* the guest whenever an IO occurred on the emulated device, on this way *QEMU* could take as long as necessary to process the IO without the guest noticing the delay. We wanted to keep the mechanism as efficient and flexible as possible, so we implemented it using the *Virtio* framework and KVM, so no emulation of the CPU was made.

To evaluate the proposed solution, we presented results from several experiments that benchmarked the emulator. The experiments revealed that the time dilation mechanism works properly, but time distortion on CPU-bound applications occurred. The behavior of the emulated device was close to the real one, the mean was 7% lower than the SSD, on average for both read and write IOs. As future work we plan to extend our technique to simulate other devices such as NVRAM devices. The code of the emulator is available for free under GPL license on [3].

ACKNOWLEDGMENT

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.

REFERENCES

[1] I. Shah, Bhavik (Ahmedabad, "Methods and systems for performing a read ahead operation using an intelligent storage adapter," April 2016. [Online]. Available: http://www.freepatentsonline.com/9311021.html

[2] J. Flower and K. Gajjar, "Adaptive intelligent storage controller and associated methods," Feb. 9 2016, uS Patent 9,256,542.

[3] "Emulator source code," https://github.com/AndrePZiviani/qemu-ssd-emulator.

[4] Y.-C. Lee, C.-T. Kuo, and L.-P. Chang, "Design and implementation of a virtual platform of solid-state disks," *IEEE Embedded Systems Letters*, vol. 4, no. 4, pp. 90–93, 2012.

[5] Z. Gu and Q. Zhao, "A state-of-the-art survey on real-time issues in embedded systems virtualization," *"Journal of Software Engineering and Applications"*, vol. 5, no. 4, pp. 277–290, 2012.

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

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

[8] A. Gordon, N. Har'El, A. Landau, M. Ben-Yehuda, and A. Traeger, "Towards Exitless and Efficient Paravirtual I/O," in *Proceedings of the 5th Annual International Systems and Storage Conference*, ser. SYSTOR '12. New York, NY, USA: ACM, 2012, pp. 10:1–10:6. [Online]. Available: http://doi.acm.org/10.1145/2367589.2367593

[9] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[10] T. Y. Kim, D. H. Kang, D. Lee, and Y. I. Eom, "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)*, 2015.

[11] Z. Amsden, "Timekeeping Virtualization for X86-Based Architectures," http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/virtual/kvm/timekeeping.txt, 2010, accessed in 15/04/2015.

[12] S. Lee, K. Fleming, J. Park, K. Ha, A. Caulfield, S. Swanson, . Arvind, and J. Kim, "BlueSSD: An Open Platform for Cross-layer Experiments for NAND Flash-based SSDs," in *WARP - 5th Annual Workshop on Architectural Research Prototyping*, O. Hammami and S. Larrabee, Eds., Saint Malo, France, Jun. 2010. [Online]. Available: https://hal.inria.fr/inria-00494143

[13] J. Lee, E. Byun, H. Park, J. Choi, D. Lee, and S. H. Noh, "CPS-SIM: Configurable and Accurate Clock Precision Solid State Drive Simulator," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, ser. SAC '09. New York, NY, USA: ACM, 2009, pp. 318–325. [Online]. Available: http://doi.acm.org/10.1145/1529282.1529351

[14] H. Jung, S. Jung, and Y. H. Song, "Architecture exploration of flash memory storage controller through a cycle accurate profiling," *IEEE Transactions on Consumer Electronics*, vol. 57, no. 4, 2011.

[15] T. Lu, P. Huang, X. He, and M. Zhang, "Understanding the Impact of Cache Locations on Storage Performance and Energy Consumption of Virtualization Systems," in *USENIX Workshop on Cool Topics on Sustainable Data Centers (CoolDC 16)*, 2016.

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

[17] S. Hajnoczi, "Coroutines in QEMU: The basics," http://blog.vmsplice.net/2014/01/coroutines-in-qemu-basics.html, 2014, accessed in 02/03/2015.

[18] J. Axboe, "blktrace user guide," http://www.cse.unsw.edu.au/aaronc/iosched/doc/blktrace.html, 2007, accessed in 16/11/2016.

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

[20] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *ACM SIGARCH Computer architecture news*, vol. 41, no. 3. ACM, 2013, pp. 475–486.